For Reference

NOT TO BE TAKEN FROM THIS ROOM

Ex libris
universitatis
albertaensis

QUAECUMQUE VERA

THE UNIVERSITY OF ALBERTA

PORTABLE EFFICIENT ALGORITHMS FOR GRAPH THEORETIC PROBLEMS

by

© YUNG HYANG TSIN

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL 1983

To the memory of my dear mother Wu Kiew (1916-1981),

to my dear father and brothers,

to my dear love, Lotus (Veronica Berrios).

ABSTRACT

This thesis focuses on the design and analysis of portable efficient algorithms for graph theoretic problems. The aim is to gain a deeper insight into the nature of parallel computation; in particular concerning the time and hardware resource tradeoffs as well as the portability of algorithms among computer models. The class of problems investigated are the following: finding the lowest common ancestors for vertex pairs of a directed tree; finding all fundamental cycles of an undirected graph, determining a directed spanning forest of an undirected graph; solving the two colorability, bridge-connectivity, bridge-connectivity augmentation and biconnectivity problems of an undirected graph. For the PRAM(Parallel RAM), it is shown that all these algorithms achieve the $O(\lg^2 n)$ time bound ($\lg n$ denotes $\lceil \log_2 n \rceil$ and n is the size of the vertex set), with the first two algorithms using $n\lceil n/\lg n \rceil$ processors and the remaining algorithms using $n\lceil n/\lg^2 n \rceil$ processors. With the exception of the first two algorithms, these results are optimal with respect to the time-processor product for dense graphs. It is also shown that for any probability error ϵ, where $0<\epsilon<1$, these algorithms could run in probabilistic $O(\lg n)$ time using $n^3|E|\lg n$ processors, where E is the edge set of the undirected graph. The performance of these algorithms when running on an abstract model is also analyzed. It is shown that they require the same amount of hardware resources and at most a factor of $\max(\lg d, \lg d'')+1$, $1 \le d, d'' \le n$, more time

than the ordinary matrix multiplication algorithm on the abstract model (d and d'' are diameters). This result immediately implies that all these algorithms could achieve the $O(n)$ time bound on the MCN (Mesh-connected Networks), the $O(\lg^2 n)$ time bound on the PSN (Perfect Shuffle Networks), CCC (Cube-connected Cycles), OTN (Orthogonal Tree Networks), OTC (Orthogonal Tree Cycles), SIMD-CCC (SIMD · Cube-connected Computers) and the $O(\lg n)$ time bound on the WRAM model using at most n^4 processors. The expected time complexity of these algorithms is also discussed. It is shown that with the exception of the last two problems, all the algorithms have expected time $O(\lg n \cdot \lg\lg n)$ on the PSN, CCC, OTN, OTC, SIMD-CCC and the PRAM and have expected time $O(\lg\lg n)$ on the WRAM. It is also shown that for the conventional sequential computer model, the biconnectivity and bridge-connectivity algorithms could run in optimal time and space.

A general program scheme for finding the bridges of an undirected graph is also presented. It is shown that by substituting various specific functions for the parameters in the program scheme, a number of optimal algorithms for finding the bridges can be derived. Included in these are the known optimal sequential algorithms and new optimal parallel algorithms for finding the bridges.

The possibility of breaching the $O(\lg^2 n)$ time bound is also examined. It is shown that for the recognition problems of split graphs and permutation graphs, $O(\lg n)$ deterministic

time algorithms can be deduced from good characteristic theorems.

Acknowledgements

I would like to express my gratitude and appreciation to my thesis supervisor Dr. Francis Chin for introducing me to this interesting field, and for his guidance and patience. I would also like to thank the members of my thesis committee, Drs. Harvey Abbott, Stanley Cabay, Len Schubert and Howard Siegel for their helpful comments. In particular, I thank Dr. Howard Siegel, my external examiner, for taking the trouble to make a long trip to Edmonton.

I am also grateful to the Department of Computing Science for providing me with various forms of financial assistance during the last few years. Special thanks go to Dr. Dennis Ward for helping me to improve the readability of this thesis.

Table of Contents

Chapter Page

1. INTRODUCTION 1

   1.1 Background 1

   1.2 Thesis Outline and Main Results 6

   1.3 Definitions and Notations 10

2. EFFICIENT ALGORITHMS FOR THE PRAM 16

   2.1 Introduction and Previous Results 16

   2.2 Preliminary Results 17

      2.2.1 Two Useful Lemmas 17

      2.2.2 Finding All Paths from the Vertices to the Roots in an Inverted Forest 18

   2.3 Constructing a Directed Spanning Forest in an Undirected Graph 21

   2.4 Finding the Lowest Common Ancestors of q Vertex Pairs in a Directed Tree 30

   2.5 Finding all Fundamental Cycles of an Undirected Graph 32

   2.6 2-coloring an Undirected Graph 34

   2.7 Finding the HLCA(u)'s 36

      2.7.1 Motivation and Definition 36

      2.7.2 Computing the HLCA(u)'s Based on Preorder Numbering 39

      2.7.3 Computing the Preorder Numbers 41

      2.7.4 Conclusions 44

   2.8 The Bridge-connectivity Problem 44

      2.8.1 Introduction 44

      2.8.2 Finding All the Bridges in an Undirected Graph 45

      2.8.3 The Bridge-connected Components of an Undirected Graph 47

2.9 The Bride-connectivity Augmentation Problem 48

2.10 The Biconnectivity Problem 51

2.10.1 Introduction 51

2.10.2 Finding all Biconnected Components in an Undirected Graph 52

2.10.3 Finding all the Separation Vertices in an Undirected Graph 57

2.11 Conclusions 59

3. IMPLEMENTATION ON THE MMM MODEL 63

3.1 Introduction 63

3.2 The Computer Model MMM 64

3.2.1 Definitions 64

3.2.2 Some Perliminary Results 67

3.3 Constructing a Breadth-first Search (Directed) Spanning Forest of an Undirected Graph 70

3.4 Finding the Lowest Common Ancestors of all Vertex Pairs in a Directed Tree 72

3.5 Finding a set of Fundamental Cycles of an Undirected Graph 74

3.6 2-coloring an Undirected Graph 75

3.7 The Bridge-connectivity Problem 77

3.8 The Bridge-connectivity Augmentation Problem 79

3.9 The Biconnectivity Problem 87

3.10 Performance on Existing Models 89

3.11 Conclusions 93

4. IMPLEMENTATION ON THE SEQUENTIAL RAM 96

4.1 Introduction 96

4.2 The Sequential Algorithm for Biconnectivity 97

4.2.1 An Outline of the Algorithm 98

4.2.2 Partitioning the Directed Tree 99

4.2.3 Combining the Trimmed-subtrees 104

4.2.4 Discussion of Other Related Work 112

4.3 A General Program Scheme for Finding Bridges ..113

4.3.1 The General Program Scheme 113

4.3.2 Implementation on the Sequential RAM ...114

4.3.3 Implementation on the PRAM 117

4.4 Conclusions 118

5. PROBABILISTIC TIME, EXPECTED TIME AND O(lgn) TIME COMPLEXITIES 119

5.1 Introduction 119

5.2 Probabilistic Time Complexity 120

5.3 Expected Time Complexity 124

5.4 O(lgn) Time Algorithms for Split Graphs and Permutation Graphs 125

5.5 Identification of Split Graphs 126

5.6 Identification of Permutation Graphs 134

5.7 Conclusions 135

6. CONCLUSIONS 137

Bibliography 145

APPENDIX : Some Detailed Implementations 153

List of Figures

Figure Description Page

2.1 A directed tree and its array $\mathbf{F}^+$ 22

2.2 Adjusting the edges of 1-tree-loops to form an inverted tree 25

2.3(i) $G(V,E)$ 28

2.3(ii) A potential inverted spanning tree of G 28

2.3(iii) An inverted spanning tree of G 29

2.4 An illustration of HLCA(u) 38

4.1(i) A directed spanning tree $T(V,E')$ 101

4.1(ii) The partition $\{T_i\}$ of T 102

4.1(iii) The partition $\{BC_j\}$ of T 103

4.2(i) Non-tree edges violating the — definition 107

4.2(ii) Non-tree edges violating the — definition 107

4.2(iii) Non-tree edges violating the — definition 107

5.1 A split graph 128

Chapter 1

INTRODUCTION

1.1 Background

The advances in device technology over the past decade have contributed an enormous increase in the speed of computation. However, as the speed of computer devices reach their ultimate physical limitations, system performance in the future can only be significantly improved through parallelism. This has stimulated much of the research activity on parallel computation during the past decade. Since the basic role of computers is to carry out computation, the design of efficient algorithms for various classes of problems is always desirable. As a result, research in this area has been very active. In this thesis, our concern is graph theoretic problems.

Graph theoretic problems arise naturally in many contexts. For instance, scheduling in operations research, analyzing networks and designing potential circuit boards in electrical engineering, designing reliable networks for communication, identifying isomorphic structures in chemical compounds and investigating the fine structures of the gene, etc., can all be conveniently formulated in terms of graphs. Due to the widespread applications of graphs, the design of efficient algorithms for graph theoretic problems is of both theoretical and practical interest. For the conventional sequential computer, an enormous number of papers devoted to

efficient graph algorithms have been published over the last twenty years. By contrast, there were few such algorithms for parallel computer models until the mid-seventies when several $O(\lg^2 n)$ time[1] parallel algorithms for the graph-connectivity and transitive closure problems appeared. Since then, the design of efficient algorithms for graph theoretic problems on parallel computer models has drawn a great deal of interest. However, despite these efforts, *efficient* parallel graph algorithms are still comparatively rare.

Of the parallel algorithms published in the literature, most are designed for the SIMD shared memory model, allowing read conflicts but not write conflicts. Recently, the name PRAM (Parallel RAM) was attached to this model[WYLL79, BORO82] and has been widely accepted. Briefly speaking, the PRAM has an unlimited number of sequential RAM's all of which have access to a common memory of unlimited size (we shall call these sequential RAM's "processors" henceforth). Each processor is assigned a unique positive integer called the processor index. At any time, several processors may read the same memory location at the same time, but at no time may more than one of them write into the same memory location. The processors are synchronized and operated under the control of a single instruction stream propagated by a control unit. There is also an enable/disable mask which can be used to prevent a subset of the processors from executing

---

[1] $\lg n$ stands for $\lceil \log_2 n \rceil$ and n is the size of the vertex set of the undirected graph.

an instruction.

As opposed to the conventional sequential computer, one has to account for the amount of hardware resources used when one designs algorithms for a parallel computer model. The hardware resources are measured in terms of the number of processors, or the size of the chip area if VLSI technology is employed. This makes the situation more complicated, as there are now three resources — time, memory space, hardware resource — for which one has to account. Partly because the relationships between these three resources are not well understood yet, and partly because of memory space is cheap compared to time and hardware resources, researchers have always ignored the space resources (unless they are unreasonably large) and concentrated on minimizing the amount of time and hardware resources (in particular, the number of processors) used in designing parallel algorithms.

From the description of PRAM, it is not difficult to perceive that there is a close relationship between sequential RAM and PRAM. Let A be an algorithm designed for a problem P on the PRAM. If A takes $T(n)$ time and $P(n)$ processors for an instance of P of size n, then given a sequential RAM, the sequential RAM can simulate execution of algorithm A on the PRAM by executing every instruction of A $P(n)$ times. At the ith repetition, the sequential RAM will behave exactly like the ith processor of the PRAM when the PRAM is executing A. Clearly, it would take a total of

$T(n) \cdot P(n)$ time for the sequential RAM to complete the execution of algorithm A. An implication of this observation is that $T(n) \cdot P(n) \geq Ł(n)$, where $Ł(n)$ is the lower bound for problems of size n on the sequential RAM.

Let $G(V,E)$ be a graph where $|V|=n$.[2] There are two data structures which have been widely used to represent G on the sequential RAM. These are the adjacency list and adjacency matrix[TARJ72,EVEN79]. If an adjacency list is used to represent G, then it is well-known that $Ł(n)=\Omega(n+|E|)$. However, adjacency lists seem to be inappropriate for SIMD computers. A more appropriate data structure which has been widely used is the adjacency matrix, and throughout this thesis, we shall use adjacency matrix to represent graphs on parallel computers. For graph theoretic problems concerning non-trivial monotone[3] graph properties, it has been proven that if the input graph is represented by an nxn adjacency matrix, then $Ł(n)=\Omega(n^2)$[KIRK74,RIVE76]. Moreover, it is easily shown that for non-trivial graph theoretic[4] problems, $\Omega(\lg n)$ is a lower bound for $T(n)$ on the PRAM[SAVA77]. As a consequence, $P(n) \geq \lceil n^2/\lg n \rceil$ on the PRAM for achieving the $O(\lg n)$ time bound for non-trivial graph theoretic problems if the adjacency matrix is used as input data structure. In other words, in designing parallel graph algorithms on the

---

[2] All graph-theoretic terms are defined in Section 1.3

[3] A graph property is **non-trivial** if there are some graphs possessing the property and some which do not. A graph property is **monotone** if whenever a graph $G(V,E)$ possesses the property, then any graph $G'(V,E')$ where E is a subset of E' also possesses the property.

[4] A graph theoretic problem is non-trivial if at least one of its output is a function of all its input.

PRAM, the $O(\lg n)$ time and $\lceil n^2/\lg n\rceil$ processors bound is the best one can possibly achieve if adjacency matrices are used to represent graphs. Up to the present, no one has managed to achieve the $O(\lg n)$ time bound on the PRAM. The best time bound achieved so far is $O(\lg^2 n)$, and there is strong evidence that $O(\lg^2 n)$ may be a lower bound for time on the PRAM, although no proof has been given. Consequently, the more promising optimal bounds one could achieve on the PRAM are the $O(\lg^2 n)$ time and $\lceil n^2/\lg^2 n\rceil$ processors bounds.

Many of the graph theoretic problems do have parallel algorithms achieving the $O(\lg^2 n)$ time bound. However, the number of processors used to achieve this time bound is always greater than $O(n^2/\lg^2 n)$. The only exception is the graph-connectivity problem and some of its equivalent problems. The first parallel algorithm for this problem running on the PRAM achieved the $O(\lg^2 n)$ time bound with n^3 processors[ARJO75,REGB78]. The processor bound was then improved to n^2 independently by Hirschberg[HIRS76] and Savage[SAVA77] and to $O(|E|+n\lg n)$ by Ja'Ja'[JAJA78] (E is the edge set of the given graph). Hirschberg, Chandra and Sarwate[HIRS79] further improved the processor bound to $O(n^2/\lg n)$ and Wyllie improved it to $n+|E|$[WYLL79]. Finally, Chin, Lam and Chen[CHIN81,CHIN82] managed to improve the bound to $O(n^2/\lg^2 n)$ (note that Ja'Ja' and Wyllie's algorithms have $T(n)\cdot P(n)\not\geq\Omega(n^2)$. This does not give rise to a contradiction to our previous discussion, because they did not use adjacency matrices to represent graphs. Their

results are not optimal for either sparse graphs or dense graphs). Parallel algorithms for other graph theoretic problems which achieve the $O(\lg^2 n)$ time bound but with a greater number of processors in the PRAM can be found in [ARJO75,ATAL82,CHAN76,GOLD77,JAJA78,JAJA82,REGH78,SAVA77, SAVA81]. Others which run in $\Omega(n)$ time can be found in [ARJO75,ECKS77a,ECKS77b,REGB78,SHIL81,VISH81a].

The PRAM has received the most attention in the past decade but has also received criticism for its impracticability for construction by current technology. In view of this, some researchers began to design graph algorithms for other more restrictive models which can be constructed with current technology. Apparently, designing graph algorithms on these models is much more difficult than on the PRAM. Up to the present time, only a few algorithms for some basic graph theoretic problems (mainly for the graph-connectivity problem) have been reported for a few of these models[NASS81,NATH81,NATH82,ATAL82,AWER83].

1.2 Thesis Outline and Main Results

In this thesis, we focus on the design and analysis of efficient algorithms for a class of graph theoretic problems on various computer models. This class of problems includes the following: finding the lowest common ancestors for vertex pairs of a directed tree; finding all fundamental cycles of an undirected graph, determining a directed spanning forest of an undirected graph, solving the

two-colorability, bridges connectivity, bridge-connectivity augmentation and biconnectivity problems of an undirected graph, and recognizing splits graphs and permutation graphs. This class of problems has drawn a great deal of interest recently and efficient algorithms for sloving them on various computer models have been developed[ATAL82,SAVA81, REIF82a,REIF82b].

Traditionally, whenever an algorithm is presented, it is designed with a particular model in mind, and its complexity analysis is provided for that model only. There are at least two drawbacks with this approach. Firstly, it is difficult to compare two different algorithms for the same problem if they are designed for different computer models. Secondly, extra effort has to be made in order to carry it over to other models. A typical example is Hirschberg's graph-connectivity algorithm which was originally designed for the PRAM. It was then implemented on the MCN (Mesh-connected Networks) by Nassimi and Sahni[NASS81]; on the PSN (Perfect Shuffle Networks) by Schwartz[SCHW80], on the WRAM by Shiloach and Vishkin[SHIL82a,VISH82]; and finally on the PSN (Perfect Shuffle Networks), OTN (Orthogonal Tree Networks), and OTC (Orthogonal Tree Cycles) by Nath, Maheshwari and Bhatt[NATH81,NATH82]. It would be convenient if the complexity analysis of an algorithm could be given in such a way that it would be valid for *any* model satisfying certain moderate conditions.

In this thesis, we shall design efficient algorithms which are portable in the sense that they can run efficiently on many computer models. In particular, they run on an abstract model, called *MMM*, which includes a large class of parallel computer models as special cases.

In the next section, definitions are provided for terms and notations to be used in subsequent chapters.

In Chapter 2, efficient algorithms are presented for the class of graph-theoretic problems listed above except the last two problems on the PRAM. All these algorithms achieve the $O(\lg^2 n)$ time bound, with the first two algorithms using $n\lceil n/\lg n\rceil$ processors and the remaining algorithms using $n\lceil n/\lg^2 n\rceil$ processors. In all cases, our algorithms are better than the best previously known algorithms and in most cases reduce the number of processors used by a factor of $n\lg n$. Moreover, our algorithms are optimal with respect to the time-processor product for dense graphs with the exception of the first two algorithms.

In Chapter 3, it is shown how the algorithms presented in Chapter 2 could be implemented efficiently on other more restrictive SIMD models. This is accomplished by first proposing an abstract model, called *MMM*, which satisfies certain moderate constraints and then implementing the algorithms on the *MMM*. It is shown that most of these algorithms achieve the $O(\lg^2 n)$ time bound with $\lceil n^3/\lg n\rceil$ processors on the PRAM and many restrictive SIMD models; the $O(\lg n)$ time bound with n^3 processors on the WRAM (a stronger

PRAM), and the $O(n)$ time bound with n^2 processors on the Mesh-connected Networks.

In Chapter 4, the implementation of these algorithms on the conventional sequential model is explored. It is shown that the biconnectivity algorithm can be implemented on the sequential computer in optimal time and space. Moreover, the algorithm is shown to be a generalization of the best previously known sequential algorithm for the same problem. The bridge-connectivity algorithm is also generalized to a general program scheme for finding the bridges in an undirected graph. This general program scheme includes most of the best previously known sequential algorithms as special cases. In addition to that, new parallel algorithms, including the one presented in Chapter 2, can be deduced from it.

In Chapter 5, the possibility of breaching the $O(\lg^2 n)$ time bound is examined. Based on some of the recent results due to Reif[REIF82a,REIF82b], it is shown that given any probability error ϵ, $0<\epsilon<1$, our algorithms could run in $O(\lg n)$ time using $|E|n^3\lg n$ processors on the PRAM with probability less than ϵ that an error will occur. It is also shown that the expected time complexity for most of the algorithms described in Chapter 3 is $O(\lg n \cdot \lg\lg n)$ on the PSN, CCC, OTN, OTC, SIMD-CCC and PRAM and is $O(\lg\lg n)$ on the WRAM. The recognition problems for split graphs and permutation graphs are also studied and $O(\lg n)$ deterministic time algorithms are presented.

Finally, in Chapter 6, our results are summarized and some open problems for further research are listed.

1.3 Definitions and Notations

A **graph** $G(V,E)$ consists of a finite non-empty set V of **vertices** and a set E of pairs of vertices called **edges.** Without loss of generality, we assume $V=\{1,2,...,n\}$ throughout this thesis. If the edges are unordered pairs, then G is **undirected;** otherwise G is **directed.** $G(V,E)$ is **sparse** if $|E|=O(n)$ and is **dense** if $|E|=O(n^2)$. For undirected graphs, an edge joining the vertices a and b is represented by (a,b). Furthermore, (a,b) and (b,a) are considered as identical elements. For directed graphs, an edge from vertex a to vertex b is represented by $<a,b>$. a is called the **tail** of the edge while b is called the **head** of the edge. The **underlying graph** of a directed graph $G'(V,E')$ is an undirected graph $G(V,E)$ such that $(u,v)\epsilon E$ iff $<u,v>$ or $<v,u>\epsilon E'$. A graph $G'(V',E')$ is a **subgraph** of a graph $G(V,E)$ if V' is a subset of V and E' is a subset of E. Let V' be a subset of V. The graph $G'(V',V'\text{x}V'\Lambda E)$ is called a subgraph of $G(V,E)$ **induced by** V' (Λ stands for set intersection here). An **adjacency matrix** M of an undirected (resp. directed) graph $G(V,E)$ is a $n\text{x}n$ Boolean matrix such that $M[u,v]=1$ iff $(u,v)\epsilon E$ (resp. $<u,v>\epsilon E$).

Let $P=\{u_0,u_1,...,u_k\}$ be a sequence of vertices of an undirected graph $G(V,E)$, P is called a **walk** in G if $(u_i,u_{i+1})\epsilon E$, $0\leq i<k$. We say that (u_i,u_{i+1}) is an **edge on P.**

The **length** of P is k. A **path** is a walk in which $u_i \neq u_j$ for $i \neq j$. A **cycle** is a walk in which $u_1 = u_k$ and no edge in G appears more than once. A **simple cycle** is a path in which $u_1 = u_k$. **Directed walks, directed paths, directed cycles** and **directed simple cycles** are defined in the similar way.

Let $G(V,E)$ be an undirected graph; if for every two vertices u,v in V, there is a path in G joining u and v, then G is **connected.** Each connected maximal subgraph of G is called a **connected component** of G. The **diameter** of G is the length of the longest minimal path between all vertex pairs if G is connected and is the longest diameter of all the connected components of G if G is disconnected. Diameters for directed graphs can be defined in a similar way. Let v be a vertex in G. The **degree** of v is the number of edges in G incident on v. If the degree of v is 0, then v is called an **isolated vertex**; if the degree of v is 1, then v is called a **pendant.** If all vertices in G have the same degree, then G is a **regular** graph.

A **tree** is a connected undirected graph with no cycles in it. Let $T(V',E')$ be a directed graph. T is said to have a **root** r, if $r \in V'$ and every vertex $v \in V'$ is reachable from r via a directed path. If the underlying undirected graph of T is a tree, then T is a **directed tree.** If, moreover, the underlying graph of T is a subgraph of a connected undirected graph $G(V,E)$ such that $V'=V$, then T is a **directed spanning tree** in G. A **directed forest** is a graph whose connected components are directed trees. If T is a directed

forest such that each directed tree in T is a directed spanning tree of a connected component of an undirected graph G and vice versa, then T is called a **directed spanning forest** of G. If the edges of T are all reversed, the resulting graph is called an **inverted spanning forest** of G. **Inverted spanning trees, inverted trees, inverted forests** etc. are defined similarly. Let $<a,b>$ (resp. $<b,a>$) be an edge in a directed (resp. inverted) tree. a is the **father** of b and b is a **son** of a. Let c, d be any two vertices in a directed (resp. inverted) tree T, c is an **ancestor** of d if $c=d$ or there exists a directed path from c to d (resp. from d to c) in T. c is a **proper ancestor** of d if c is an ancestor of d and $c \neq d$. d is a **descendant** of c if c is an ancestor of d.

Throughout this thesis, we denote the 'undirected' path from vertex a to vertex b in a (directed) tree by $[a{*}{\rightarrow}b]$, and by $[a{*}{\rightarrow}b)$ if vertex b is to be excluded. If the path consists of at least one edge, then the '*' is removed from the notation. Moreover, we denote $u \mathrel{\dot{\leq}} v$ iff u is an ancestor of v in the tree and $u \lessdot v$ iff u is a proper ancestor of v. Let $T(V,E')$ be an inverted (directed) spanning forest of an undirected graph $G(V,E)$. The graph $G-T$ is an undirected graph whose vertex set is V and whose edge set is $E-\{(u,v) \mid <v,u> \in E'\}$. Any edge in $G-T$ is called a **non-tree edge**. To simplify our notation, we shall use $E-E'$ to denote the edge set of $G-T$. Let $G_1(V_1,E_1)$ and $G_2(V_2,E_2)$ be two graphs. $G_1 U G_2$ is a graph whose vertex set is $V_1 U V_2$ and the

edge set is $E_1 \cup E_2$. $G_1 \Lambda G_2$ is a graph whose vertex set is $V_1 \Lambda V_2$ and the edge set is $E_1 \Lambda E_2$ (Λ stands for set intersection here). $G_1 \uplus G_2$ is a graph whose vertex set is $V_1 \cup V_2$ and the edge set consists of edges that are either in E_1 or E_2, but *not* in both.

An inverted (directed) tree T is called an **ordered tree** if the sons of every vertex in T are ordered. If v is the ith son of a vertex in T, then the **rank** of v is i.

The preorder and postorder traversals of an inverted (directed) tree are defined as follows:

Preorder traversal

(i) Visit the root of the tree.

(ii) Traverse each subtree of the root in preorder, in order of rank.

Postorder traversal

(i) Traverse each subtree of the root in postorder, in order of rank.

(ii) Visit the root of the tree.

Note that there is no inorder traversal for trees as there is no obvious place to insert a root among its descendants. If in the course of traversing an ordered tree in preorder, vertex v is the kth vertex visited, then the **preorder number** of v is defined to be k. **Postorder number** can be defined similarly.

Let $T(V',E')$ be a directed tree, and $u,v \in V'$. The **lowest common ancestor LCA**(u,v) of u and v in T is the vertex $w \in V'$ such that w is a common ancestor of u and v and any other

common ancestor of u and v in T is also an ancestor of w in T. If T is a spanning tree of a connected, undirected graph G, let (u,v) be an edge in $G-T$, then the cycle in G consisting of the paths $[u *\!\!\rightarrow \mathrm{LCA}(u,v)]$, $[\mathrm{LCA}(u,v) *\!\!\rightarrow v]$ and the edge (v,u) is a **fundamental cycle** in G. An undirected graph $G(V,E)$ is **2-colorable (bipartite)** if V can be partitioned into V_1 and V_2 such that no edge in G has both of its end-vertices in V_1 or V_2. For $e \epsilon E$, e is a **bridge** in G iff e is not on any cycle in G. Let B be the set of bridges in G; then every connected component of the graph $G'(V,E-B)$ is a **bridge-connected component** of G. The **bridge-connectivity augmentation problem** is the problem of adding the minimum number of edges to a graph so as to bridge-connect the graph. For $a \epsilon V$, if there exist $u,v \epsilon V$ such that u, v, a are all distinct and that every path connecting u and v in G passes through a, then a is called a **separation vertex** of G. A graph is **biconnected** if it contains no separation vertex. Every maximal biconnected subgraph of G is called a **biconnected component** of G.

Let $G(V,E)$ be an undirected graph. G is **independant** if $E=\emptyset$ and G is **complete** if $E=V \times V$. An undirected graph $G(V,E)$ is a **split graph** iff V can be partitioned into two disjoint subsets V_1,V_2 such that the graph $G_1(V_1,E_1)$ induced by V_1 is independent and the graph $G_2(V_2,E_2)$ induced by V_2 is complete. We shall call $\{G_1(V_1,E_1),G_2(V_2,E_2)\}$ a **split** of G. A **clique** in G is a maximal complete subgraph of G.

Let $P=[P(1),P(2),P(3),\ldots,P(n)]$ be a permutation of V. Let $E(P)=\{(i,j)|P^{-1}(i)<P^{-1}(j)$ and $i>j$ or $P^{-1}(i)>P^{-1}(j)$ and $i<j,\ i,j\epsilon V\}$, where $P^{-1}(i)$ is the element in V which P maps into i. The **permutation graph** of P is the undirected graph $G(V,E(P))$. A directed graph $G'(V,E')$ is **transitive** if (i,j) and $(j,k)\epsilon E' \Rightarrow (i,k)\epsilon E'$.

Chapter 2

EFFICIENT ALGORITHMS FOR THE PRAM

2.1 Introduction and Previous Results

In this chapter, we shall present efficient algorithms for the class of graph theoretic problems listed in the Introduction except the recognition problems of split graphs and permutation graphs which will be deal with in Chapter 5. The computer model we use is the widely accepted PRAM[WYLL79]. In subsequent chapters, we will consider the implementation of these algorithms on other computer models.

The class of problems we investigate in this chapter has been studied by various people before. The best known results for the PRAM were due to Savage and Ja'Ja'[SAVA81]. They designed parallel algorithms for these problems and achieved an $O(\lg^2 n)$ time bound with the processor-time products being $O(n^2 \lg^2 n)$ for the directed spanning tree problem and being $O(n^3)$ or $O(n^2(\lg n)^m)$ where $m \geq 3$ for the remaining problems. In this chapter, the algorithm we present for the lowest common ancestors problem takes $O(\lceil q/nK \rceil . \lg n + n/K)$ time with $nK (K>0)$ processors, where q is the number of vertex pairs whose lowest common ancestors are to be found. The algorithm for the fundamental cycles problem takes $O(\lceil |E|/nK \rceil . \lg n + n/K + \lg^2 n)$ time with $nK (K>0)$ processors, where E is the edge set of the undirected graph. The algorithms for the directed spanning forest, the 2-colorability, the bridge-connectivity, the

bridge-connectivity augmentation and the biconnectivity problems all take $O(n/K+\lg^2 n)$ time with $nK (K>0)$ processors. In particular, an $O(\lg^2 n)$ time bound can be achieved with $K=\lceil n/\lg n\rceil$ for the first two problems and with $K=\lceil n/\lg^2 n\rceil$ for the remaining problems. Since the processor-time products of our algorithms are at most $O(n^2 \lg n)$, for $0<K\leq\lceil n/\lg^2 n\rceil$, our algorithms are better than Savage and Ja'Ja's' in all cases and in most cases use a factor of $n\lg n$ fewer processors. Except for the algorithms for the first two problems, the processor-time products of our algorithms are $O(n^2)$, which is optimal for dense graphs.

Besides being more efficient, our algorithms also assume bounded parallelism as opposed to the unbounded parallelism adopted by Savage, Ja'Ja' and many others. Bounded parallelism is more realistic as it can cope with the situation where the number of processors available is smaller than the input size.

Throughout this chapter, we assume that the input to each algorithms is an adjacency matrix, and the arithmetic operations, +, - as well as the boolean operations each takes one time unit to execute.

2.2 Preliminary Results

2.2.1 Two Useful Lemmas

In this section, we list two lemmas which will be used frequently in analyzing the time and processor complexities

in this chapter.

Lemma 2.1: Given n elements $\{a_0, a_1, \ldots , a_{n-1}\}$, let f be a function to be applied to every element. If computing $f(a_i)$ takes t time units and $K(\geq 1)$ processors are provided, then $f(a_i)$, $0 \leq i \leq n-1$, can be computed in $\lceil n/K \rceil \cdot t$ parallel time units.

Lemma 2.2:[CHIN81,CHIN82] Given n elements $\{a_0, a_1, \ldots , a_{n-1}\}$ and K processors, $A(n)=a_0*a_1*a_2*\ldots*a_{n-1}$ can be computed in T parallel time units where $*$ is any associative binary operator and

$$T = \lceil n/K \rceil - 1 + \lg K \quad \text{if } \lfloor n/2 \rfloor > K$$
$$= \lg n \quad \text{if } \lfloor n/2 \rfloor \leq K$$

2.2.2 Finding All Paths from the Vertices to the Roots in an Inverted Forest

In this section, we present a method for constructing an array, denoted by $\mathbf{F}^+$, in which each row contains a path from a vertex to a root in an inverted forest. The array will be very useful in the design of parallel algorithms presented in the following sections.

Let $T(V',E')$ be an inverted forest with $|V'|=n$. Without loss of generality, we assume $V'=\{1,2,\ldots,n\}$. Let $\{T_j\}$ be the set of all inverted trees in T and $\{r_j\}$ be the set of all their roots.

Definition : F : $V' \to V'$ is a function such that

$F(i)$ = the father of the vertex i in T for $i \notin \{r_j\}$;

$F(r) = r$, $\forall r \in \{r_j\}$.

The function F can be represented by a directed graph F which can be constructed from T by adding a self-loop at each root r_j in T.

From the function F, we define $F^k, k \geq 0$, as follows:

Definition : $F^k: V' \to V'$, $k \geq 0$, is a function such that

$F^0(i) = i$, $\forall i \in V'$;

$F^k(i) = F(F^{k-1}(i))$, $\forall i \in V', k > 0$.

If i is a vertex in T_j, $F^k(i)$ is the kth ancestor of i in T_j or r_j.

Definition : For each $i \in V'$, if i is in T_j, for some j, then $depth(i) = \min\{k \mid F^k(i) = r_j$ and $0 \leq k \leq n-1\}$.

The concepts $F^k(i)$, $k \geq 0$, and $depth(i), 1 \leq i \leq n$, were first introduced by Savage in [SAVA77]. It was shown that given the function F of a directed forest T (T could be a directed forest or its inverted forest), $F^k(i)$, $0 \leq k \leq n-1$, and $depth(i)$, $1 \leq i \leq n$, can be computed in $O(\lg n)$ time with n^2 processors and $n\lceil n/\lg n\rceil$ processors respectively. In the following, we will show in Theorem 2.3 that $F^k(i)$, $0 \leq k \leq n-1$, $1 \leq i \leq n$, can indeed be computed in $O(\lg n)$ time with $n\lceil n/\lg n\rceil$ processors or in $O(\lg^2 n)$ time with $n\lceil n/\lg^2 n\rceil$ processors and then $depth(i)$ in $O(\lg n)$ additional time with n processors.

Theorem 2.3: (i) Given the function F of a directed or an

inverted forest T, $F^k(i), 1 \le i \le n, 0 \le k \le n-1$ can be computed in $O(n/K+\lg n)$ time with $nK (K>0)$ processors on a PRAM. (ii) Given $F^k(i)$, $1 \le i \le n$, $0 \le k \le n-1$, and $nK (K>0)$ processors, $depth(i), 1 \le i \le n$ can be computed in $O(\lg(n/K))$ time if $K \ge 1$ or in $O(\lceil 1/K \rceil \cdot \lg n)$ time if $0<K<1$ on a PRAM.

Proof: To compute F^k, for all $0 \le k \le n-1$, we proceed in two steps:

```
1. for $i: 1 \le i \le n$ pardo $F^0(i) := i$; $F^1(i) := F(i)$ dopar;
2. for $t := 0$ to $\lg(n-1)-1$ do
      for $s: 1 \le s \le 2^t$, $i: 1 \le i \le n$ pardo
           $F^{2**t+s}(i) := F^{2**t}(F^s(i))$
      dopar;
```

If nK processors are given, it is clear that step 1 can be computed in $O(\lceil 1/K \rceil)$ time(Lemma 2.1). Step 2 can be computed in[5] $\sum_{t=0}^{\lg(n-1)-1} (\lceil 2^t/K \rceil)$

$= \lg K + \sum_{t=\lg k}^{\lg(n-1)-1} (\lceil 2^t/K \rceil)$

$< \lg K + \lg(n-1) - \lg K + 1/K \sum_{t=\lg k}^{\lg(n-1)-1} 2^t$

$= O(n/K+\lg n)$ time units.

Once $F^k(i), 1 \le i \le n, 0 \le k \le n-1$, are computed, $depth(i), 1 \le i \le n$, can be found by performing a binary search on the ordered sequence $F^0(i), F^1(i), \ldots\ldots, F^{n-1}(i)$, for each i, searching for the left-most occurrence of r_j using $F^{n-1}(i) (=r_j)$ as the key. This takes a total of $O(\lceil 1/K \rceil \cdot \lg n)$ time units if $0<K<1$. For $K \ge 1$, the search is performed in the following way: divide the sequence into $\lceil n/K \rceil$ segments, assign one processor to each segment and perform simultaneously a binary search to search for the left-most occurrence of r_j in each segment. After this step, every processor compares

---

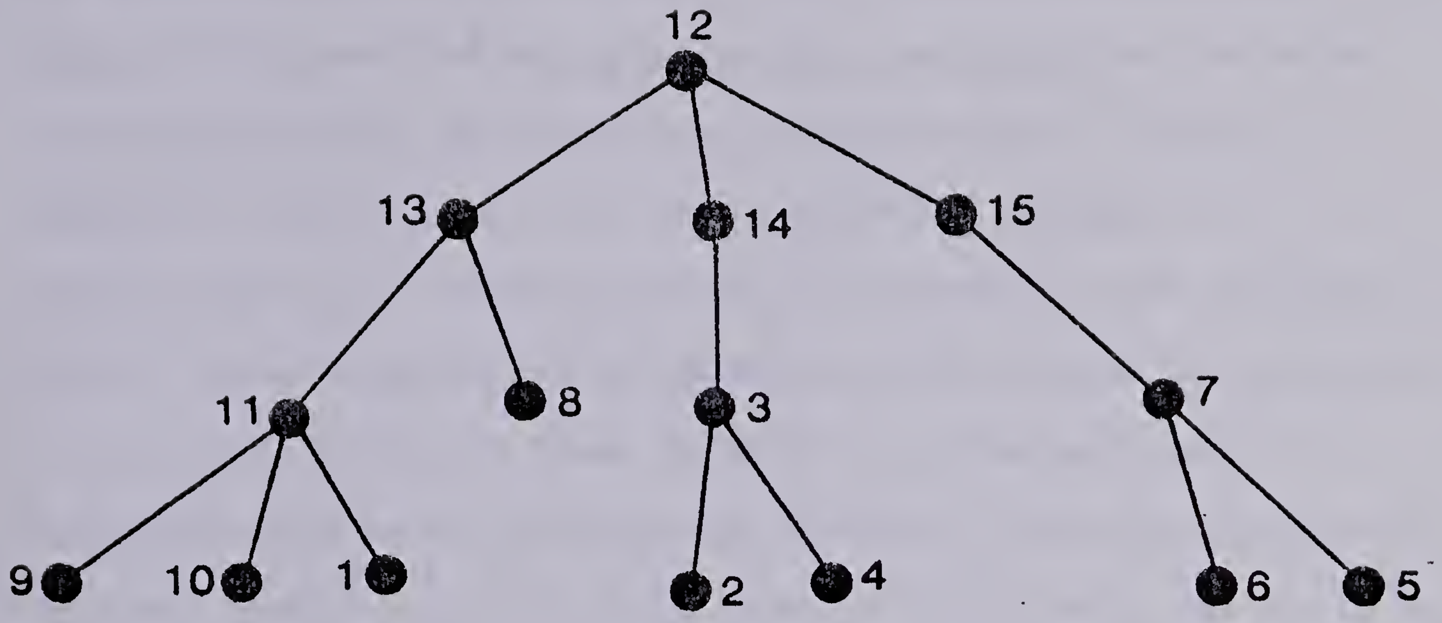
[5] Due to the limitation of our character set, we must use lġ to represent lg in superscripts and subscripts.

the element it finds with the preceding and succeeding elements in the sequence. There is exactly one processor which does not have all the three elements distinct or identical and this processor locates the left-most occurrence of r_j in the sequence. This takes a total of $O(\lg\lceil n/K\rceil)$ time units. ■

The actual computations of $F^k(i), 1 \leq i \leq n, 0 \leq k \leq n-1$, and $depth(i), 1 \leq i \leq n$, are performed in an array $\mathbf{F}^+$ in which $\mathbf{F}^+[i,k]$ contains $F^k(i)$. After the computations are finished, each row of $\mathbf{F}^+$ is right shifted so that all the r_j's except the left-most one are eliminated. As a consequence, the right-most column of the array contains only the roots from $\{r_j\}$. Furthermore, for each vertex i, all occurrences of i appear only in column $(n-1)-depth(i)$. For each row i, a number, $n+i$, acting as an undefined value, is inserted into the first $(n-1)-depth(i)$ entries. These adjustments are done for convenience and not out of necessity and they take $O(n/K)$ time with $nK (K>0)$ processors(Lemma 2.1). The adjusted array, $\mathbf{F}^+$, of an inverted tree is depicted in Figure 2.1. Note that the ith row in $\mathbf{F}^+$ contains the path from vertex i to a root in T.

2.3 Constructing a Directed Spanning Forest in an Undirected Graph

In this section, we present an efficient parallel algorithm for constructing a directed spanning forest in an undirected graph $G(V,E)$. In view of the fact that it is the



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	16	16	16	16	16	16	16	16	16	16	16	1	11	13	12
2	17	17	17	17	17	17	17	17	17	17	17	2	3	14	12
3	18	18	18	18	18	18	18	18	18	18	18	18	3	14	12
4	19	19	19	19	19	19	19	19	19	19	19	4	3	14	12
5	20	20	20	20	20	20	20	20	20	20	20	5	7	15	12
6	21	21	21	21	21	21	21	21	21	21	21	6	7	15	12
7	22	22	22	22	22	22	22	22	22	22	22	22	7	15	12
8	23	23	23	23	23	23	23	23	23	23	23	23	8	13	12
9	24	24	24	24	24	24	24	24	24	24	24	9	11	13	12
10	25	25	25	25	25	25	25	25	25	25	25	10	11	13	12
11	26	26	26	26	26	26	26	26	26	26	26	26	11	13	12
12	27	27	27	27	27	27	27	27	27	27	27	27	27	27	12
13	28	28	28	28	28	28	28	28	28	28	28	28	28	13	12
14	29	29	29	29	29	29	29	29	29	29	29	29	29	14	12
15	30	30	30	30	30	30	30	30	30	30	30	30	30	15	12

Figure 2.1 A directed tree and its array F^+.
Note that since $n=15$, any number greater than 15 serves as an undefined value in the array.

inverted spanning forest of G which is useful in the design of other parallel algorithms in the following sections, the algorithm presented below actually constructs an inverted spanning forest. Nevertheless, converting an inverted spanning forest into a directed spanning forest is straightforward. This algorithm will serve as the backbone of the other algorithms presented in the following sections. It takes $O(n/K+\lg^2 n)$ time if $nK (K \geq 1)$ processors are available and could achieve the $O(\lg^2 n)$ time bound using the optimal number of processors. The previous best result takes n^2 processors to achieve the $O(\lg^2 n)$ time bound[SAVA77].

This algorithm is based on the algorithm for finding an undirected spanning forest presented in [CHIN82] and the array $\mathbf{F}^+$ presented in the last section. The latter is used to assign a direction to each edge in the undirected spanning forest generated by the former.[6]

We first give a general description for the strategy used in our algorithm. In the course of running the algorithm for finding an undirected spanning forest[CHIN82], a number of *1*-tree-loops [HIRS79][7] are generated. Each of these *1*-tree-loop is a directed graph whose vertices are supervertices generated during the previous iteration (a **supervertex** is a vertex in G or a *1*-tree-loop). The edges of these *1*-tree-loop will be included in the undirected

---

[6] We assume the reader is familiar with the undirected spanning forest algorithm. For those who are not, we refer them to reference [CHIN82].

[7] A *1*-tree-loop is a directed graph in which every vertex has outdegree 1 and in which there is exactly one cycle and the length of the cycle is 2.

spanning forest and all these edges are directed edges whose directions are ignored by the algorithm in [CHIN82]. If the only loop in a *1*-tree-loop is destroyed by eliminating the out-going edge from the smallest-numbered-vertex, the resulting graph is an inverted tree. As a result, when the loops of all the *1*-tree-loops are destroyed in this way, the resulting graph (built by embedding the modified (acyclic) *1*-tree-loops created during one iteration into the modified (acyclic) *1*-tree-loops created during the following iteration) may well be an inverted spanning forest. Unfortunately, this is not the case in general because some vertices may result in having two fathers. This situation is depicted in Figure 2.2, where a directed edge $<a,b>$ is selected during iteration $j+1$ to connect two supervertices S_1 and S_2 created during iteration j. The two graphs resulting from the two supervertices are inverted trees. However, since a is not the root r_1 of S_1, a will have two fathers after S_1 and S_2 have been included into a single supervertex. Therefore, the graph $S_1 U S_2$ is not an inverted tree, by definition, unless the directions of all the edges on the path from a to r_1 are reversed. The same situation occurs in $S_2 U S_3$ when the directed edge $<c,d>$ is selected to connect S_2 and S_3. To overcome this difficulty, we have to reverse the directions of all edges on the path from a to r_1 and those on the path from c to r_2. The array $\mathbf{F}^+$, described in Section 2, contains the path from any vertex to a root in an inverted forest T; hence we can generate the array $\mathbf{F}^+$

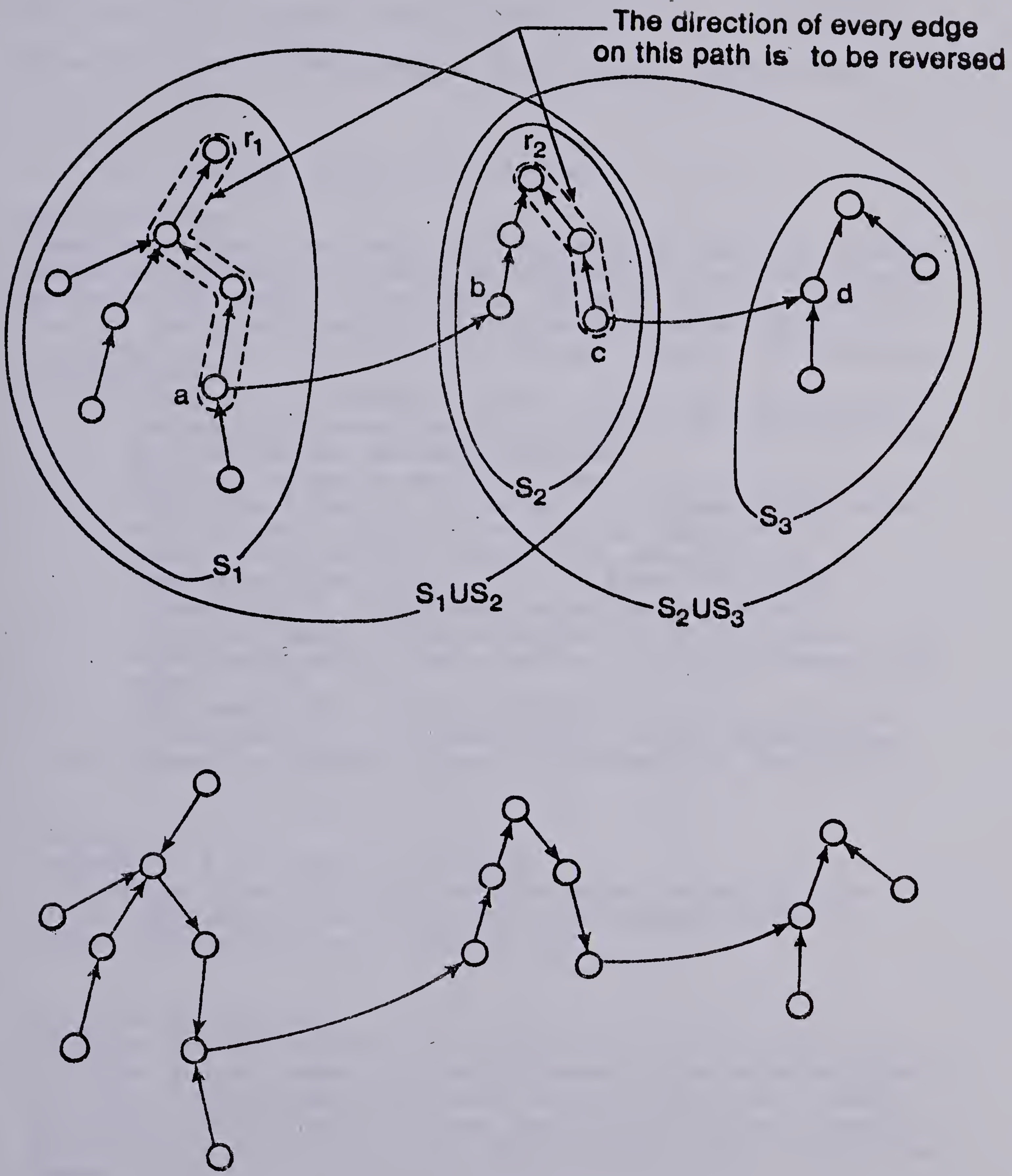

The direction of every edge on this path is to be reversed
r_1
r_2
b
a
c
d
S_1
S_2
S_3
S_1US_2
S_2US_3

Figure 2.2 Adjusting the edges of 1-tree-loops to form an inverted tree

covering both S_1 and S_2. By retrieving the ath row and the cth row of $\mathbf{F}^+$, we can identify the set of all edges whose directions are to be reversed in S_1 and S_2 respectively.

Our algorithm runs in two stages.

Algorithm DSF

Stage 1: (* The first stage is basically a modified version of the algorithm for finding an undirected spanning forest. We refer the reader to reference [CHIN82] for the details.*) Execute the algorithm for finding an undirected spanning tree; during each iteration j, $1 \le j \le \lg n$, record the following information:

a. Convert the forest of all 1-tree-loops generated during this iteration into a forest of inverted trees by eliminating the edge from the smallest-numbered-vertex of each 1-tree-loop and store the forest in a vector F_j. (*Note: This vector acts as the function F defined in Section 2.*)

b. Record the 'actual' edges in G establishing the connection specified in F_j. (* Note: The edges recorded in F_j are *pseudo* edges which connect 'supervertices'. They do not exist in G. However, for each pseudo edge, there exists a corresponding actual edge in G.*)

c. The vector $D[1..n]$ generated during this iteration is stored as D_j. (* Note: $D_j[v]$ is the supervertex containing vertex v when iteration j is completed.*)

Stage 2:

1. Generate $\mathbf{F}^+_j$'s from F_j, $1 \le j \le \lg n$.
2. (* Adjust the directions of the edges, starting from those recorded during iteration $\lg n$, gradually down to those recorded during iteration 1.*)

```
$R' := \{v \in V \mid D_{\lg n}[v] = v\}$;
(* Note: In the following for loop, $R'$ contains the tails
of those actual edges in $G$ which connect two supervertices
in the inverted trees generated during iteration $i$, where
$j < i \le \lg n$. It includes all those vertices which have two or
more fathers in the directed graph formed upon the inverted
trees *).
for $j := \lg n$ downto 1 do
    begin
    i) For every $r' \in R'$,
         reverse the direction of every 'pseudo' edge lying
         on the path from the supervertex $D_j[r']$ to the
         root of the inverted tree, in $F_j$, containing
```

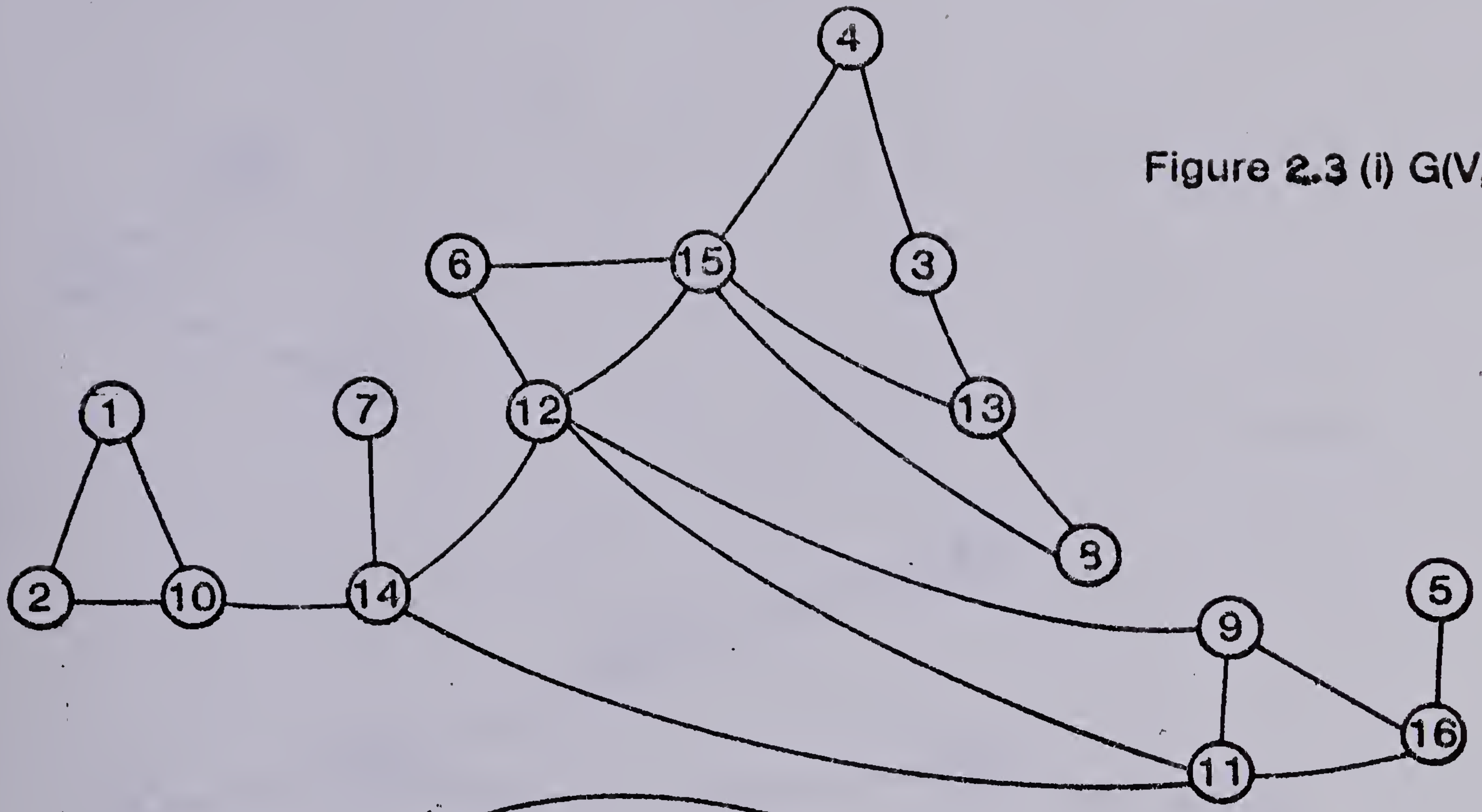
```
        $D_j[r']$;
    ii) Output all the 'actual' edges in $G$ corresponding to
        the pseudo edges in $F_j$;
    iii) $R'=R' \cup \{v \in V | v$ is the tail of an 'actual' edge
              output in step ii)$\}$
    end; ■
```

A complete example is given in Figure 2.3 and a detailed implementation using the method described above is given in the Appendix.

Theorem 2.4: Algorithm DSF correctly generates an inverted spanning forest for an undirected graph.

Proof: (Backward induction) In Stage 1, an inverted forest F_j is correctly generated during each iteration $j, 1 \leq j \leq \lg n$[CHIN82]. In Stage 2, supposing that after processing $F_i, j \leq i \leq \lg n$, an inverted forest F'_j is created. Clearly, F'_j and F_j must have the same vertex set V_j. When processing F_{j-1}, it should be clear that there exists a one to one correspondence between the vertices in V_j and the inverted trees in F_{j-1}. This implies that no two instances of r' in R' will belong to the same inverted tree in F_{j-1}. As a result, after Step 2 i), each inverted tree in F_{j-1} is effectively modified so as to root at the supervertex $D_{j-1}[r']$. These modified inverted trees are then embedded into the inverted forest F'_j in Step 2 ii), the resulting directed graph F'_{j-1} is clearly an inverted forest. But $F'_{\lg n}=F_{\lg n}$ is an inverted forest initially, therefore by induction, F'_1 must be an inverted forest and hence an inverted spanning forest for G.■

Figure 2.3 (i) G(V,E)


4
6
15
3
1
7
12
13
8
2
10
14
9
5
16
11

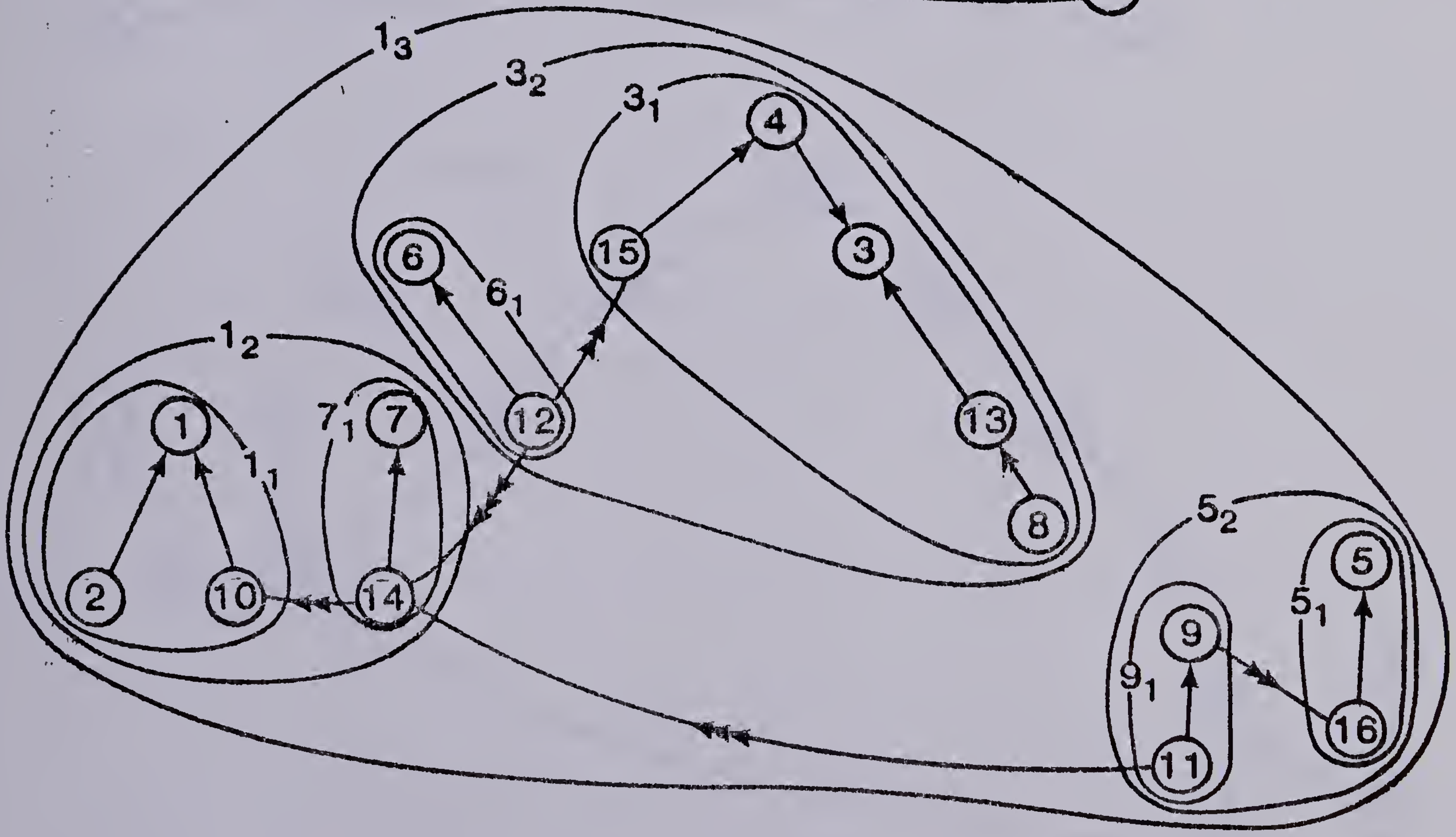

1₃
3₂
3₁
4
6
15
3
6₁
1₂
1
7₁
7
12
13
1₁
8
5₂
2
10
14
5
9
5₁
9₁
16
11

Figure 2.3(ii) A potential inverted spanning tree of G.

——► a directed edge selected during the first iteration;
——►► a directed edge selected during the second iteration;
——►►► a directed edge selected during the third iteration.

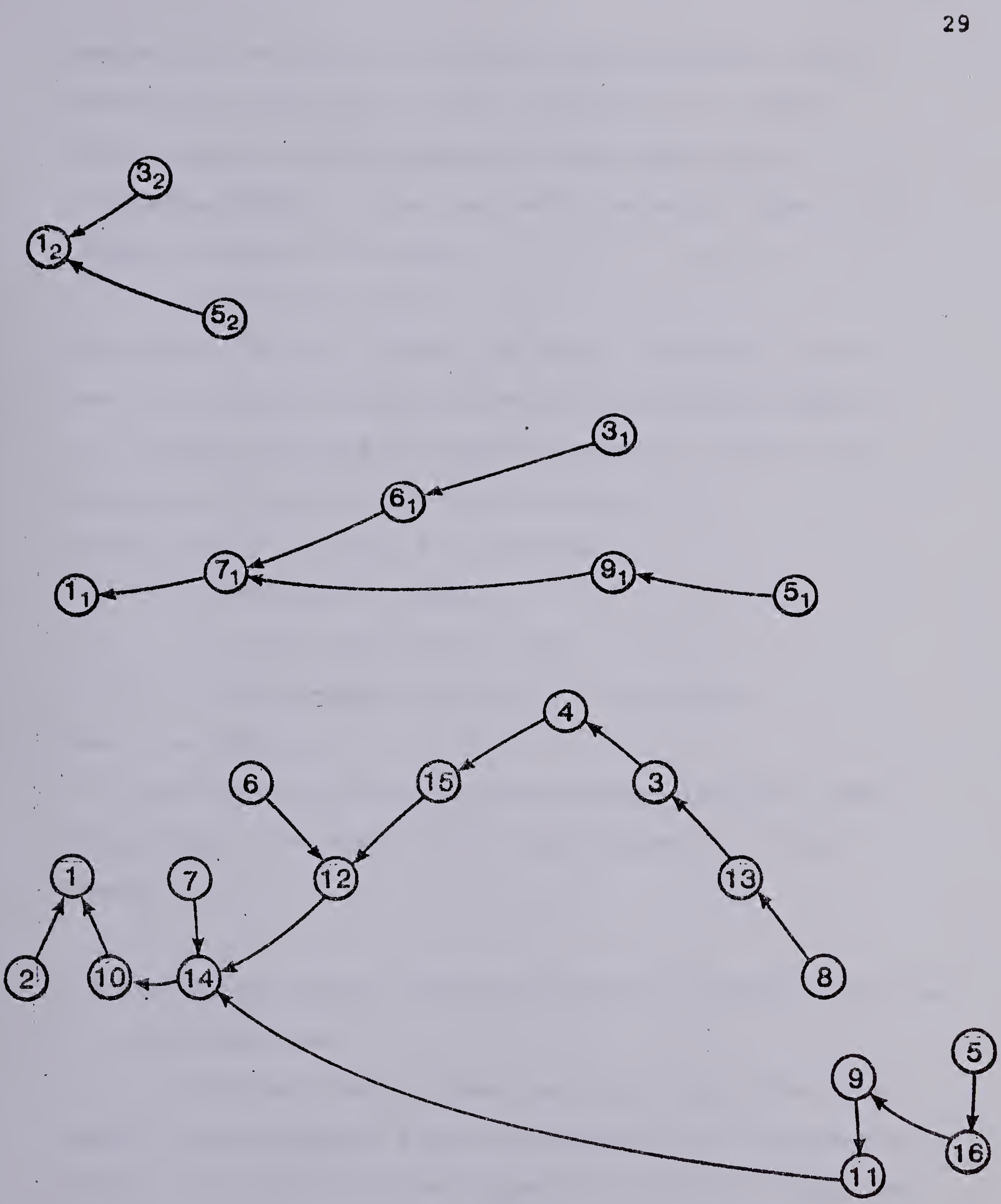

3_2
1_2
5_2
3_1
6_1
7_1
1_1
9_1
5_1
4
6
15
3
1
7
12
13
2
10
14
8
5
9
16
11

Figure 2.3(iii) An inverted spanning tree of G.

Theorem 2.5: Finding an inverted spanning forest takes $O(n/K+\lg^2 n)$ time with nK $(K\geq 1)$ processors on a PRAM.

Proof: Stage 1 takes $O(n/K+\lg^2 n)$ time with $nK(K\geq 1)$ processors[CHIN82]. Since the total number of edges in the inverted forests is at most

$$\Sigma_{j=1}^{\lg n}\lfloor n/2^{j-1}\rfloor<2n,$$

the creation of $\mathbf{F}_j^\dagger$, $1\leq j\leq \lg n$, in Step 1 of Stage 2 can be done in $O(n/K+\lg n)$ time with $nK(K>0)$ processors (Theorem 2.3). Steps 2 ii) and iii) each takes $O(1)$ time for each iteration. Since the size of $\mathbf{F}_j^\dagger, 1\leq j\leq \lg n$, is $\lceil n/2^{j-1}\rceil \times \lceil n/2^{j-1}\rceil$, Step 2 i) requires

$$\Sigma_{j=1}^{\lg n}\lceil\lceil n/2^{j-1}\rceil^2/nK\rceil$$

$$< \lg n + \Sigma_{j=1}^{\lg n}\lceil n/2^{j-1}\rceil^2/nK$$

$$= O(n/K+\lg n) \text{ time for } \lg n \text{ iterations.}$$

Hence the theorem.■

Note that the processor-time product is $O(n^2)$, when $1\leq K\leq\lceil n/\lg^2 n\rceil$, the algorithm is thus optimal for dense graphs.

2.4 Finding the Lowest Common Ancestors of q Vertex Pairs in a Directed Tree

As with the inverted spanning forest algorithm, the lowest common ancestor algorithm presented in this section plays a key role in the development of parallel algorithms for other graph theoretic problems to be discussed in the following sections. The previous best algorithm was due to Savage and Ja'Ja'[SAVA81]. Their algorithm first computes

the transitive closure of the adjacency matrix of the directed tree, and then uses the transitive closure to determine the set of all common ancestors of every vertex pair. The $\mathbf{min}_{\leq}$ operation is then applied over each set of common ancestors to determine the lowest common ancestors for all the vertex pairs. Since there are at worst $O(n^2)$ vertex pairs and each takes $n/2$ processors to evaluate the $\mathbf{min}_{\leq}$ operator, this algorithm requires $O(n^3)$ processors to achieve the $O(\lg^2 n)$ time bound.

In this section, we shall show that we can combine the array $\mathbf{F}^+$ described in Section 2.2.2 and the binary search technique to develope a new algorithm for the lowest common ancestor problem which takes at worst n^2 processors to achieve the $O(\lg n)$ time bound.

Let $T(V',E')$ be a directed tree and $V'=\{1,2,\ldots,n\}$. Let a and b be a pair of vertices and c is their lowest common ancestor; then row a and row b of $\mathbf{F}^+$ will have identical contents between column $(n-1)$-$depth[c]$ and column $n-1$, inclusive, and will have different contents in the other columns. As a result, to determine c, we can perform a binary search on row a and row b simultaneously in the following way: if the two entries being examined in row a and row b (in the same column, of course) are different, the search is continued on the right-half, otherwise it is continued on the left-half. It takes $O(\lg n)$ time units to find c with one processor. In general, we have:

Theorem 2.6: Given q vertex pairs, $1 \leq q \leq n^2$, finding the lowest common ancestors for these vertex pairs takes $O(\lceil q/n\mathrm{K} \rceil . \lg n + n/\mathrm{K})$ time on a PRAM if $n\mathrm{K}(\mathrm{K}>0)$ processors are available.

Proof: Constructing the array $\mathbf{F}^{+}$ takes $O(n/\mathrm{K}+\lg n)$ time (Theorem 2.3) and finding the lowest common ancestors of the q vertex pairs takes $\lceil q/n\mathrm{K} \rceil . \lg n$ time units, if $n\mathrm{K} \leq q \leq n^2$ (Lemma 2.1) or $\lg n+1$ time units, if $n\mathrm{K}>q$. Thus finding the lowest common ancestors of $q(1 \leq q \leq n^2)$ vertex pairs, takes $O(\lceil q/n\mathrm{K} \rceil . \lg n + n/\mathrm{K})$ time with $n\mathrm{K}(\mathrm{K}>0)$ processors.■

A detailed description of this algorithm is given in the Appendix (see Algorithm LCA). In particular, when $\mathrm{K}=n$ and $\lceil n/\lg n \rceil$, the lowest common ancestors can be found in $O(\lg n)$ and $O(\lg^2 n)$ time respectively.

2.5 Finding all Fundamental Cycles of an Undirected Graph

Without loss of generality, we assume that the undirected graph $G(V,E)$ is connected from this section onwards, unless otherwise stated.

It is known that a set of fundamental cycles of a connected, undirected graph $G(V,E)$ can be determined from a spanning tree $T(V,E')$ of G [REIN77]. Specifically, let $LCA(a,b)$ be the lowest common ancestor of a and b in T and (a,b) is an edge in $G-T$, then (a,b) together with the paths $[b \ast\!\!\rightarrow LCA(a,b)]$ and $[LCA(a,b) \ast\!\!\rightarrow a]$ form a fundamental cycle.

Based on the above observation, we can easily find a set of fundamental cycles of G as follows:

First, an inverted spanning tree T of G is found, using the algorithm presented in Section 2.3 which takes $O(n/\mathrm{K}+\lg^2 n)$ time with $n\mathrm{K}(\mathrm{K}\geq 1)$ processors. Algorithm LCA (see Appendix) is then called to determine the lowest common ancestor for every pair of vertices (a,b) in G-T. The algorithm returns the ordered pair $(\mathbf{LCA}^+,\mathbf{F}^+)$ and the vector *depth*, where $\mathbf{LCA}^+[a,b]$ contains the lowest common ancestor of (a,b). A vector $\mathbf{P}^+$ is then created such that $\mathbf{P}^+[v]$ contains the value $(n-1)-depth[v]$ which is the column number of v in $\mathbf{F}^+$. Hence, for each (a,b) in G-T, the path from column $\mathbf{P}^+[a]$ to column $\mathbf{P}^+[\mathbf{LCA}^+[a,b]]$ in row a and the path from column $\mathbf{P}^+[b]$ to column $\mathbf{P}^+[\mathbf{LCA}^+[a,b]]$ in row b of $\mathbf{F}^+$ and the edge (a,b) determine a fundamental cycle in G.

The correctness of the algorithm is easily verified. Since the number of vertex pairs $q=|E|-|E'|$, the algorithm obviously takes $O(\lceil |E|/n\mathrm{K}\rceil . \lg n+n/\mathrm{K}+\lg^2 n)$ time with $n\mathrm{K}(\mathrm{K}\geq 1)$ processors. In particular, the $O(\lg^2 n)$ time bound is achieved with $\mathrm{K}=n/\lg n$. Note that the output of the algorithm are stored in an $O(n^2)$ compact data structure, which consists of the triple $(\mathbf{P}^+,\mathbf{LCA}^+,\mathbf{F}^+)$.

At this point, it is interesting to note that the best sequential algorithm for the fundamental cycle problem has time complexity $O(n^3)$[REIN77]. Our algorithm presented here immediately implies a sequential algorithm having time complexity $O(n^2\lg n)$. While our performance is better, we do not intend to claim that it is an improvement over the previous result. This is because the output data structures

of the two sequential algorithms are substantially different. However, for cases where the fundamental cycle algorithm is used as an internal routine, our algorithm will be better as it requires less time and space.

2.6 2-coloring an Undirected Graph

No previous work was reported for this problem except [JAJA82] in which the $O(\lg^2 n)$ time and n^2 processors complexities were mentioned. However, the description of the algorithm was not given. In this section, we shall present an efficient algorithm which achieves the $O(\lg^2 n)$ time bound using only $n\lceil n/\lg^2 n\rceil$ processors. We first prove a lemma.

Lemma 2.7: An undirected graph $G(V,E)$ is 2-colorable(bipartite) iff it has no fundamental cycles of odd length.

Proof: The 'only if' part is immediate from the well-known property of bipartite graphs, namely an undirected graph is bipartite iff it has no cycle of odd length.

Let G has no fundamental cycles of odd length and C be any cycle in G. There exists a set of fundamental cycles Γ such that $C=\uplus\Gamma$[REIN77]. Consider two fundamental cycles C_1 and C_2 in Γ. Let $C'=C_1\uplus C_2$ and $\ell(C_i)$ denotes the length of C_i. Clearly, $\ell(C')=\ell(C_1)+\ell(C_2)-2*\ell(C_1\Lambda C_2)$, where Λ denotes 'set intersection' here. Since $\ell(C_1)$, $\ell(C_2)$, $2*\ell(C_1\Lambda C_2)$ are all even, $\ell(C')$ has to be even. A simple induction will reveal that $C=\uplus\Gamma$ is an even cycle. ■

From Lemma 2.7, we immediately have:

Corollary 2.8: Let T be an inverted spanning tree of G. G is 2-colorable(bipartite) iff for any edge e in $G-T$, one end vertex of e must be of even depth while the other is of odd depth.

Our algorithm is based on Corollary 2.8. The input to the algorithm is an adjacency matrix of the undirected graph $G(V,E)$. First, an inverted spanning tree T of G is constructed. A flag is then associated with every vertex pair in $V \times V$. This flag is set to **true** initially. Then for every non-tree edge (u,v) in $G-T$, the condition : "Is one of the depths of u, v odd while the other is even?" is tested. If the answer is negative, then the associated flag will be set to **false**. After this step, all the flags are **anded** together. G is bipartite iff the result is **true.** If G is bipartite, then the vertex set V is partitioned into V_1 and V_2. This can be accomplished by sorting the set of ordered pairs $\{(\text{'}depth(v)\text{ is odd'},v) | v \in V\}$.

Algorithm Bipartite:

```
1. Construct an inverted spanning tree $T$ for $G(V,E)$.
2.(i) for all $(u,v) \in V \times V$ pardo $flag[u,v]$:=true dopar;
  (ii) for all $(u,v)$ in $G-T$ pardo
         $flag[u,v]$:=($depth[u]$ is odd)$\wedge$($depth[v]$ is even)
                        $\vee$ ($depth[u]$ is even)$\wedge$($depth[v]$ is
                        odd)
         dopar;
3.(i) $Bipartite$:=$\wedge_{i,j} flag[i,j]$
  (ii) if $Bipartite$ then
       begin
         $V_1$:={$v | depth[v]$ is even};
```

```
    $V_2$:={v|depth[v] is odd}
  end; ■
```

Theorem 2.9 Algorithm Bipartite takes $O(n/K+\lg^2 n)$ time with $nK(K\geq 1)$ processors on a PRAM.

Proof: With $nK(K\geq 1)$ processors, Step 1 takes $O(n/K+\lg^2 n)$ time(Theorem 2.5). By Lemma 2.1, Steps 2(i) and 2(ii) take $O(n/K)$ time. Step 3(i) takes $O(n/K+\lg K)$ time units (Lemma 2.2). Step 3(ii) takes at most $O(\lg n\cdot\lg\lg n)$ time[BORO82]. The theorem thus follows.■

2.7 Finding the HLCA(u)'s

2.7.1 Motivation and Definition

In the following sections, the set of fundamental cycles of *G* plays an important role in developing optimal algorithms for the bridge-connectivity and biconnectivity problems. As a result, the efficiencies of these algorithms rely on how well we can manipulate the fundamental cycles. To prevent any fundamental cycle from being considered excessively, we associate with each of them exactly two vertices and consider it only at those two vertices. These two vertices are determined as follows: let *T* be an inverted spanning tree on which the fundamental cycles are generated and *C* be any of the fundamental cycles. The two vertices associated with *C* are the end-vertices of the non-tree edge determining *C*. With this strategy, every fundamental cycle is considered exactly twice.

Let u be any vertex in G. We find the *highest* vertex in T which can be reached from u via a fundamental cycle associated with u. This vertex is clearly an ancestor of u. Furthermore, all the edges on the closed path from u to this vertex are guaranteed to lie within the same fundamental cycle while edges which lie below u or above this vertex may or may not have this property. We denote this vertex with HLCA(u)(the prefix H stands for the highest). A precise definition is given below.

Definition: Let $G(V,E)$ be an undirected graph and $T(V,E')$ be its inverted spanning tree. Let $u \in V$, **HLCA(u)**=LCA(u,v) where $(u,v) \in E-E' \cup \{(u,u)\}$ and depth(LCA(u,v))$\leq$depth(LCA(u,v')), $\forall (u,v') \in E-E' \cup \{(u,u)\}$.

Figure 2.4 illustrates HLCA(u). The solid lines and circles represent the edges and vertices of an inverted spanning tree of an undirected graph. The dotted lines represent the edges in the graph G-T emerging from a particular vertex u.

To compute HLCA(u),$\forall u \in V$, we may first use the lowest common ancestor algorithm to find LCA(u,v),$\forall (u,v) \in E-E' \cup \{(u,u)\}$ and then apply Lemma 2.2 to find HLCA(u),$\forall u \in V$. However, in doing so, we will require $O(\lceil |E-E'|/n\mathrm{K} \rceil . \lg n + n/\mathrm{K})$ time if $n\mathrm{K}(\mathrm{K}>0)$ processors are available. In this section, we show a way of finding HLCA(u),$\forall u \in V$ in $O(n/\mathrm{K} + \lg n \cdot \lg\lg n)$ time with $n\mathrm{K}(\lg n > \mathrm{K} \geq 1)$ processors or in $O(n/\mathrm{K} + \lg n)$ time with $n\mathrm{K}(\mathrm{K} \geq \lg n)$ processors.

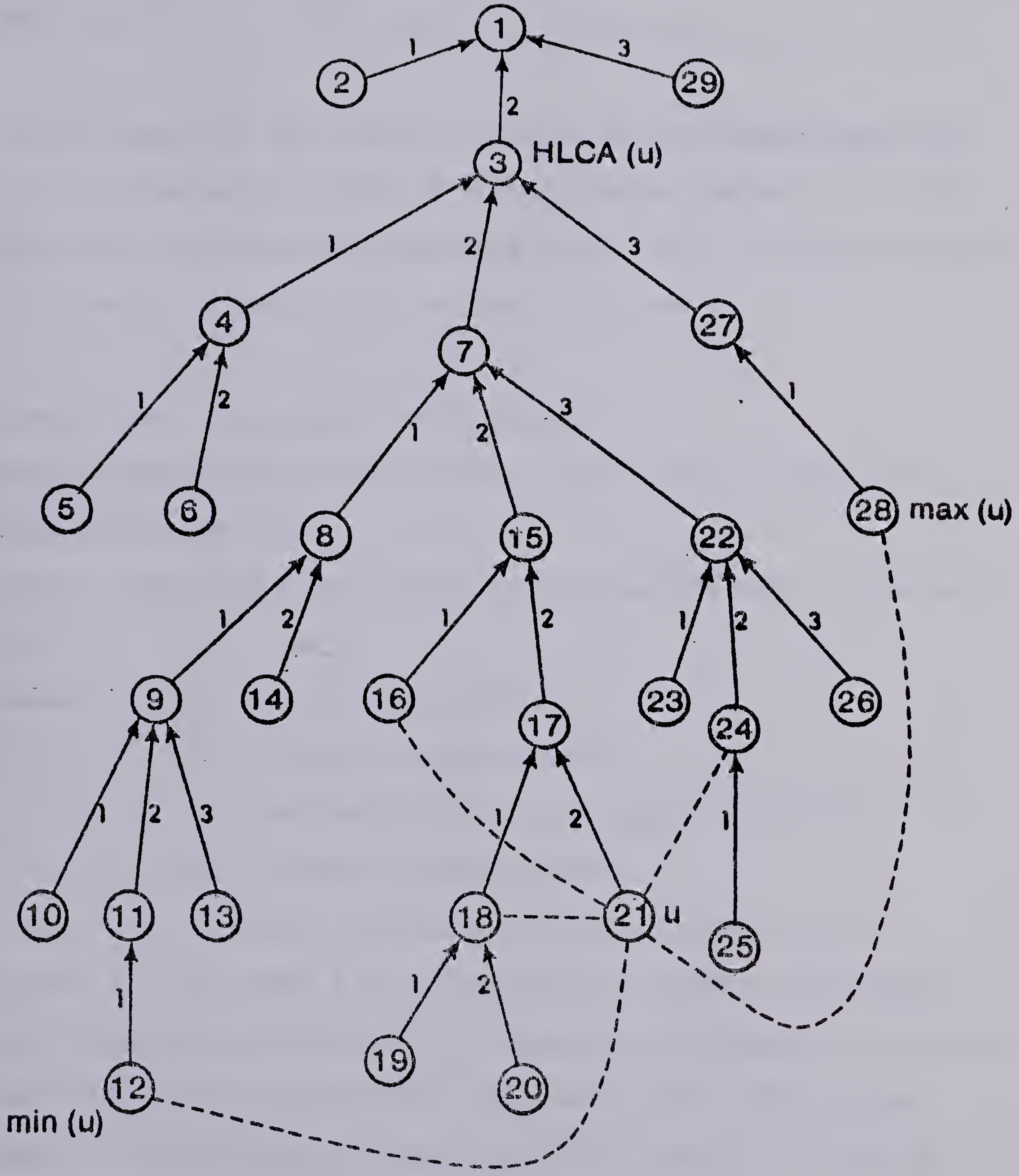

1
2
29
3 HLCA (u)
4
7
27
5
6
8
15
22
28 max (u)
9
14
16
17
23
24
26
10
11
13
18
21 u
25
19
20
12
min (u)

Figure 2.4

An illustration of HLCA(u)

This method allows us to design optimal parallel algorithms for the graph theoretic problems discussed in the following sections.

2.7.2 Computing the HLCA(u)'s Based on Preorder Numbering

The method is based on the preorder numbering of the vertices in an ordered spanning tree $T(V,E')$ of G. We denote the preorder number of a vertex v by $pre(v)$.

Lemma 2.10: Let $u,v \in V$. Then $v \preceq u$ iff $pre(v) \leq pre(u) < pre(v) + nd(v)$ where $nd(v)$ is the number of descendants of v.

Proof: Immediate from the definition of preorder traversal.■

Lemma 2.11: Let $(u,v),(u,w) \in E-E'$.

(i) If $pre(v) < pre(w) < pre(u)$,

then $\text{depth}(\text{LCA}(u,v)) \leq \text{depth}(\text{LCA}(u,w))$;

(ii) if $pre(v) > pre(w) > pre(u)$,

then $\text{depth}(\text{LCA}(u,v)) \leq \text{depth}(\text{LCA}(u,w))$.

Proof: (i) By Lemma 2.10, $pre(\text{LCA}(u,v)) \leq pre(v)$ and $pre(u) < pre(\text{LCA}(u,v)) + nd(\text{LCA}(u,v))$. Therefore $pre(\text{LCA}(u,v)) < pre(w) < pre(\text{LCA}(u,v)) + nd(\text{LCA}(u,v))$. By Lemma 2.10, $\text{LCA}(u,v) \preceq w$. Hence, $\text{depth}(\text{LCA}(u,v)) \leq \text{depth}(\text{LCA}(u,w))$. Part (ii) can be proved similarly.■

Lemma 2.11 points out that we can reduce the problem of finding HLCA(u) to that of finding the lowest common ancestor of two particular vertices in $\{v \mid (u,v) \in E-E'\} \cup \{u\}$.

Definition: Let $u \in V$, $W=\{v \mid (u,v) \in E-E'\} \cup \{u\}$.

$\mathrm{pmax}(u)=v$, where $v \in W$ and $pre(v) \geq pre(w), \forall w \in W$;

$\mathrm{pmin}(u)=v$, where $v \in W$ and $pre(v) \leq pre(w), \forall w \in W$.

Corollary 2.12:

$\mathrm{HLCA}(u)=(\min \preceq)\{\mathrm{LCA}(u,\mathrm{pmin}(u)),\mathrm{LCA}(u,\mathrm{pmax}(u))\}$.

Proof: Immediate from Lemma 2.11.■

Corollary 2.13: $\mathrm{HLCA}(u)=\mathrm{LCA}(\mathrm{pmin}(u),\mathrm{pmax}(u))$.

Proof: From Corollary 2.12, $\mathrm{HLCA}(u) \preceq \mathrm{pmin}(u)$ and $\mathrm{HLCA}(u) \preceq \mathrm{pmax}(u)$. Thus, $\mathrm{HLCA}(u) \preceq \mathrm{LCA}(\mathrm{pmin}(u),\mathrm{pmax}(u))$. By definition, $pre(\mathrm{pmin}(u)) \leq pre(u) \leq pre(\mathrm{pmax}(u))$. This implies $pre(\mathrm{LCA}(\mathrm{pmin}(u),\mathrm{pmax}(u))) \leq pre(u)$ $< pre(\mathrm{LCA}(\mathrm{pmin}(u),\mathrm{pmax}(u)))+nd(\mathrm{LCA}(\mathrm{pmin}(u),\mathrm{pmax}(u)))$. By Lemma 2.10, $\mathrm{LCA}(\mathrm{pmin}(u),\mathrm{pmax}(u)) \preceq u$. Therefore $\mathrm{LCA}(\mathrm{pmin}(u),\mathrm{pmax}(u)) \preceq \mathrm{LCA}(u,\mathrm{pmin}(u))$ and $\mathrm{LCA}(\mathrm{pmin}(u),\mathrm{pmax}(u)) \preceq \mathrm{LCA}(u,\mathrm{pmax}(u))$. By Corollary 2.12, $\mathrm{LCA}(\mathrm{pmin}(u),\mathrm{pmax}(u)) \preceq \mathrm{HLCA}(u)$. ■

Lemma 2.14: Let $T(V,E')$ be a directed tree whose vertices have been labelled in preorder; then finding $\mathrm{HLCA}(u)$, $\forall u \in V$, can be done in $O(n/\mathrm{K}+\lg n)$ time with $n\mathrm{K}(\mathrm{K} \geq 1)$ processors on a PRAM.

Proof: To compute $\mathrm{pmax}(u)$ and $\mathrm{pmin}(u)$, $\forall u \in V$, we need $O(n/\mathrm{K}+\lg \mathrm{K})$ time with $n\mathrm{K}(\mathrm{K} \geq 1)$ processors(Lemma 2.2), and to find $\mathrm{HLCA}(u)$, $\forall u \in V$, we need to find the lowest common ancestors of the n $(\mathrm{pmin}(u),\mathrm{pmax}(u))$ pairs. This takes

$O(n/K+\lg n)$ time with $nK(K>0)$ processors(Theorem 2.6).■

Figure 2.4 gives an illustration to the above lemmas and corollaries. The numbers in the circles are the preorder numbers of the vertices. For instance, the preorder number of u is 21. For convenience, we name each vertex by its preorder number. It can be easily checked that $\mathrm{depth}(\mathrm{LCA}(u,12))<\min(\mathrm{depth}(\mathrm{LCA}(u,18)),\mathrm{depth}(\mathrm{LCA}(u,16)))$, and that $\mathrm{depth}(\mathrm{LCA}(u,28))<\mathrm{depth}(\mathrm{LCA}(u,24))$. Furthermore, $\mathrm{pmin}(u)=12$, $\mathrm{pmax}(u)=28$, and $\mathrm{LCA}(12,28)=3$ which is clearly $\mathrm{HLCA}(u)$.

2.7.3 Computing the Preorder Numbers

The crucial step in computing $\mathrm{HLCA}(u),\forall u\in V$, is to determine the preorder numbers efficiently. The common way of numbering the vertices of a tree in preorder is to traverse the tree. However, this will result in an $O(n)$ time algorithm which is undesirable. In the following lemma, we show that we can carry out preorder numbering in parallel by computation rather than by traversing the tree.

Lemma 2.15: Let $T(V,E')$ be an ordered tree. For each $v\in V$,

$$pre(v)=\Sigma_s\Sigma_t nd(t)+na(v),\ s\in\mathrm{ANC}(v);\ t\in\mathrm{EBRO}(s);$$

$$=\Sigma_s nds(\mathrm{F}(s),rank(s)-1)+1+depth(v),\ s\in\mathrm{ANC}(v)-\{r\}.$$

where $\mathrm{ANC}(v)$ is the set of all ancestors of v;

$\mathrm{EBRO}(s)$ is the set of all elder brothers of s;

$nd(t)$ is the number of descendants of t;

$na(v)$ is the number of ancestors of v;

$nds(v,j)$ is the total number of descendants of the first j sons of v,

and $rank(s)$ is the rank of s, i.e. the position of s among all its brothers.

Proof: Trivial.■

Let us consider the inverted spanning tree given in Figure 2.4 again. Consider the vertex u, $pre(u)=21$, the ancestors of u are the vertices 21, 17, 15, 7, 3 and 1. The number of descendants of the elder brothers of each of these vertices except the root are 3, 1, 7, 3, and 1 respectively. These numbers sum up to 15. The number of ancestors of u is 6, this gives rise to a total sum of 21, which is the preorder number of u.

Using Lemma 2.15, we want to show that the preorder numbers $pre(v), \forall v \in V$ can be determined in $O(n/K+\lg n \cdot \lg\lg n)$ time with $nK (K \geq 1)$ processors. Assuming that an inverted tree T represented by an array $T[1..2,1..n]$ such that $\{<T[1,i],T[2,i]> \mid 1 \leq i \leq n\}=E'$ is given (Specifically, $T[1,i]=i$ and $T[2,i]=F(i)$, $1 \leq i \leq n$. We assume $T[2,r]=0$ for the root r).

Algorithm Preorder:

```
Step 1: Compute the array $\mathbf{F}^+$ and the vector $depth$ for
        $T$;
Step 2: Order the sons of every vertex in $T$, i.e.
        compute $rank(v), \forall v \in V$;
Step 3: Find $nds(v,j), \forall v \in V$, $1 \leq j \leq n(v)$, where $n(v)$ is the
        number of sons of $v$;
Step 4: Compute $pre(v), \forall v \in V$. ■
```

Lemma 2.16: Algorithm Preorder takes $O(n/K+\lg n \cdot \lg\lg n)$ time with $nK (\lg n > K \geq 1)$ processors or in $O(n/K+\lg n)$ time with

nK(K≥lg$n$) processors on the PRAM.

Proof: Step 1 can be done in $O(n/\mathrm{K}+\lg n)$ time(Theorem 2.3). In Step 2, the ordered pairs $\{<T[2,i],T[1,i]>\mid 1\leq i\leq n\}$ are sorted. This can be done in $O(\lg n\cdot\lg\lg n)$ time with n processor or in $O(\lg n)$ time with $n\lg n$ processors[BORO82]. Assuming that the sorted T is stored in $T'[1..2,1..n]$, then T' can be divided into segments such that in each segment, the first row contains the same vertex v in every entry, and the second row contains the set of all sons of v in T. The relative position of vertex i in the second row of the segment in which i resides, is the rank of i, i.e. $rank(i)$.

In step 3, $nd(v),\forall v\in V$, are first computed by scanning the $((n-1)-depth(v))$th column of $\mathbf{F}^+$ and counting the number of occurrences of v. By Lemma 2.2, this takes $O(n/\mathrm{K}+\lg\mathrm{K})$ time. After this, $nds(v,j),\forall v\in V,1\leq j\leq n(v)$, are computed using the following formula:

$$nds(v,j)=\Sigma_{i=1}^{j}nd(s_i),\ 1\leq j\leq n(v).$$

It has been shown in [KOGG73] that the partial sums $\Sigma_{i=1}^{j}a_i$, $1\leq j\leq n$, can be computed in $O(\lg n)$ time if n processors are given. Since for each vertex v, v has $n(v)$ sons, the time needed to compute $nds(v,j),1\leq j\leq n(v)$, is $O(\lg(n(v)))$ if $n(v)$ processors are assigned to v. (This is possible if we make use of the sorted array T'). As a result, all these partial sums, $nds(v,j),1\leq j\leq n(v),\forall v\in V$, can be computed in parallel in $\max\{O(\lg(n(v)))\}=O(\lg n)$ time with $\Sigma n(v)=n-1$ processors.

Finally, in step 4, $pre(v), \forall v \in V$ is computed using the formula given in Lemma 2.15. We assume $nds(v,0)=0, \forall v \in V$. Note that ANC($v$) is available in the vth row of F^+ starting from column $(n-1)$-$depth(v)$ to column $(n-1)$, and $na(v)$ equals depth(v)+1. By Lemma 2.2, this takes $O(n/K+\lg K)$ time.

Summing up, $pre(v), \forall v \in V$ can be determined in $O(n/K+\lg n \cdot \lg\lg n)$ time with $nK(\lg n>K\geq 1)$ processors or in $O(n/K+\lg n)$ time with $nK(K\geq \lg n)$ processors.■

2.7.4 Conclusions

Theorem 2.17: Computing HLCA(u), $\forall u \in V$ can be done in $O(n/K+\lg n \cdot \lg\lg n)$ time with $nK(\lg n>K\geq 1)$ processors or in $O(n/K+\lg n)$ time with $nK(K\geq \lg n)$ processors on the PRAM.

Proof: Lemmas 2.14, 2.16.■

Remark:

Since the first write-up of our algorithm for computing preorder numbers[TSIN82a], we have discovered that Schwartz described a method for computing preorder numbers on the PSN which is similar to ours[SCHW80].

2.8 The Bridge-connectivity Problem

2.8.1 Introduction

The previous best algorithm for finding the bridges in an undirected graph on the PRAM first appeared in [SAVA77]. It was then reported in [SAVA81]. This algorithm achieves

the $O(\lg^2 n)$ time bound with $n^2 \lg n$ processors. In this section, we present an optimal parallel algorithm which achieves the $O(\lg^2 n)$ time bound using only $n\lceil n/\lg^2 n\rceil$ processors.

The bridge-connectivity problem consists of two subproblems, namely finding the bridges and determining the bridge-connected components of an undirected graph. We consider the problem of finding the bridges first.

2.8.2 Finding All the Bridges in an Undirected Graph

The efficiency of our algorithm relies on the following Lemmas.

Lemma 2.18: Let $G(V,E)$ be a connected, undirected graph. If $e=(a,b)\in E$ is a bridge of G, then every inverted spanning tree of G contains either $<a,b>$ or $<b,a>$.

Proof: Trivial.■

Lemma 2.19: e is not a bridge iff e is on a fundamental cycle.

Proof: Immediate from the definition of bridges.■

The input data is again assumed to be an adjacency matrix of $G(V,E)$. By definition, an edge e is a bridge in G iff e is not contained in any cycle in G. Since there are a total of $|E|$ edges and a possible exponential number of cycles in G, basing our algorithm to find the set of all bridges on the definition may require an unmanageable number

of operations. Fortunately, thanks to Lemmas 2.18 and 2.19, we need only consider those edges in an inverted spanning tree of G and the fundamental cycles generated from that spanning tree. This allows us to start with a manageable size of edges and cycles.

Let $T(V,E')$ be an inverted spanning tree of G and $\langle a,\mathrm{F}(a)\rangle = e \in E'$. We shall show below (Theorem 2.20) that e is a bridge iff e is not included in the same fundamental cycle as any descendant of a in T. In other words, e does not lie on any of the paths $[i \ast\!\!\rightarrow \mathrm{HLCA}(i)]$ where i is a descendant of a in T. Using this characteristic of bridges, we can find all the bridges efficiently.

Theorem 2.20: Let $T(V,E')$ be an inverted spanning tree of a connected, undirected graph G, and $e = \langle a,b\rangle \in E'$.
(a,b) is a bridge of G iff for each descendant i of a, there does not exist (i,j) in $G-T$ such that
$depth(LCA[i,j]) < depth(a)$.
Proof: Let $e = \langle a,b\rangle \in E'$ be such that (a,b) is a bridge in G. If there exists (i,j) in $G-T$ such that i is a descendant of a in T and $depth(LCA[i,j]) < depth(a)$, then the path $[i \longrightarrow j \ast\!\!\rightarrow LCA[i,j] \ast\!\!\rightarrow b \longrightarrow a \ast\!\!\rightarrow i]$ is a cycle containing e. This leads to a contradiction by Lemma 2.19.
Conversely, if $e = (a,b)$ is not a bridge, then by Lemma 2.19, e is on a fundamental cycle C, i.e. there exists (i,j) in $G-T$ such that

$$\mathrm{C} : [i \longrightarrow j \ast\!\!\rightarrow LCA[i,j] \ast\!\!\rightarrow i].$$

$e \neq (i,j)$ because e is not in $G-T$ (Lemma 2.18). As a result, e is either on the path $[j \rightarrow LCA[i,j]]$ or on the path $[LCA[i,j] \rightarrow i]$, implying $depth(j) \geq depth(a) > depth(b) \geq depth(LCA[i,j])$ or $depth(i) \geq depth(a) > depth(b) \geq depth(LCA[i,j])$. Hence in either case there exists (i,j) in $G-T$ such that i is a descendant of a and $depth(LCA[i,j]) < depth(a)$.■

```
Algorithm Bridges:
1. Construct an inverted spanning tree $T(V,E')$ for $G(V,E)$.
2. Compute HLCA($u$), $\forall u \in V$.
3. Compute $\alpha(u)$, $\forall u \in V$, where
         $\alpha(u) = \min\{depth(HLCA(w)) \mid u \leq w\}$.
4. For each $<u, F(u)> \in E'$, check if depth($u$) $\leq \alpha(u)$. (* ($u$,F($u$))
   is a bridge iff depth($u$) $\leq \alpha(u)$ *) ■
```

The complexities of Algorithm Bridges is analyzed below.

Theorem 2.21: Algorithm Bridges runs in $O(n/K+\lg^2 n)$ time with $nK (K \geq 1)$ processors on a PRAM.

Proof: With $nK (K \geq 1)$ processors, step 1 takes $O(n/K+\lg^2 n)$ time (Theorem 2.5). Step 2 takes $O(n/K+\lg n \cdot \lg\lg n)$ time (Theorem 2.17). By using the array $\mathbf{F}^+$ for $T(V,E')$, Steps 3 and 4 takes $O(n/K+\lg K)$ time(Lemmas 2.1 & 2.2). Hence, algorithm Bridges runs in $O(n/K+\lg^2 n)$ time with $nK (K \geq 1)$ processors.■

2.8.3 The Bridge-connected Components of an Undirected Graph

Once the bridges of a connected, undirected graph are determined, its bridge-connected components can be

determined. Specifically, we eliminate all the bridges in G and then use Algorithm MOD.CONNECT[CHIN81,CHIN82] to find the connected components of the resulting graph. Each of the connected components thus found is a bridge-connected component of G.

The algorithm obviously runs in $O(n/K+\lg^2 n)$ time with $nK(K\geq 1)$ processors on a PRAM.

2.9 The Bride-connectivity Augmentation Problem

No previous result was reported in the literature for this problem. The algorithm presented here is a parallel version of Eswaran and Tarjan's sequential algorithm[ESWA76]. We list their algorithm below and refer the reader to the reference cited for its correctness. Note that the undirected graph G may be disconnected in this section.

```
Algorithm Brconnect[ESWA76]:
(* Given an undirected graph $G(V,E)$, add the minimum number
of edges to $G$ so that the resulting graph is
bridge-connected *)

1. Find the bridge-connected components of $G$;
2. Condense $G$ into an acyclic graph $G_0(V_0,E_0)$ by collapsing
      each bridge-connected component of $G$ into a single
      vertex;
3. Construct an edge set $A_1$ to connect the trees of $G_0$ so
      that the resulting graph $T_0(V_0,E_0 U A_1)$ is an undirected
      tree. $A_1$ is defined as follows:
           Let $\{v(i)\,|\,1\leq i\leq 2m\}$ be a set of vertices of $G_0$ such
           that
                (i) $v(2i-1)$ and $v(2i)$ are each a pendant or
                    an isolated vertex in the $i$th tree;
                (ii) $v(2i-1)=v(2i)$ iff the $i$th tree is an
                    isolated vertex.
           Then $A_1=\{(v(2i),v(2i+1))\,|\,1\leq i<m\}$;
```

4. Convert T_0 into an inverted tree for which the root has two or more sons and label its vertices with preorder numbers, then sort its pendants by preorder number;
5. Construct an edge set A_2 to bridge-connect T_0, where A_2 is defined as follows:
   Let $\{v(i) \mid 1 \le i \le p\}$ be the sorted sequence of pendants where p is the number of pendants.
   $A_2 = \{(v(i), v(i+\lfloor p/2 \rfloor)) \mid 1 \le i \le \lceil p/2 \rceil\}$;
6. Let V_1 be a set of vertices containing exactly one vertex from each bridge-connected component in G and $\pi : V_0 \rightarrow V_1$ be a 1-1 correspondence such that $\pi^{-1}(v)$ corresponds to the bridge-connected component containing v. Define $A = \{(\pi(u), \pi(v)) \mid (u,v) \in A_1 \cup A_2\}$. (* A is the minimum set of edges bridge-connecting G *). ■

The construction of the sets A_1, A_2 forms the main part of our algorithm, we handle them in the following lemmas.

Lemma 2.22: Given an adjacency matrix of $G_0(V_0, E_0)$, constructing the edge set A_1 can be done in $O(m/\mathrm{K}+\mathrm{lg}^2 m)$ time with $m\mathrm{K}(\mathrm{K} \ge 1)$ processors, where $m = |V_0|$.

Proof: First, find the connected component of G_0, i.e. compute $C(v)$, $\forall v \in V_0$ such that $C(u)=C(v)$ iff u,v belong to the same connected component in G_0. This takes $O(m/\mathrm{K}+\mathrm{lg}^2 m)$ time with $m\mathrm{K}(\mathrm{K} \ge 1)$ processors[CHIN81,CHIN82]. Then sort the set $\{<C(v),v> \mid 1 \le v \le m\}$. This takes $O(\mathrm{lg}m\,\mathrm{lglg}m)$ time with m processors[BORO82]. After that, assign one processor to each $<C(v),v>$, $1 \le v < m$, and compare the C value of that element with the C value of the following element, say $<C(u),u>$, in the sorted sequence. The processor will add (u,v) to A_1 iff $C(u) \ne C(v)$. This takes $O(1)$ time with $m-1$ processors. Hence, constructing A_1 can be done in $O(m/\mathrm{K}+\mathrm{lg}^2 m)$ time with $m\mathrm{K}(\mathrm{K} \ge 1)$ processors.■

Lemma 2.23: Given an adjacency matrix of the undirected tree

$T(V_0,E')$, the edge set A_2 can be constructed in $O(m/K+\lg^2 m)$ time with $mK(K\geq 1)$ processors, where $m=|V_0|$.

proof: Find an inverted tree T_0' of T_0 such that the root has more than one son. This takes $O(m/K+\lg^2 m)$ time with $mK(K\geq 1)$ processors(Theorem 2.5). Then label the vertices with preorder numbers and identify the pendants as follows: sort $\{<F(v),v>|1\leq v\leq m\}$. Clearly, the vertices having the same father are in consecutive positions after sorting. To avoid write conflicts, only the processor assigned to the leftmost vertex of each segment of vertices having the same father in the sorted sequence will write a 1 into an appropiate entry of an array *mark* to indicate that the father is a nonpendant. Consequently, $mark(v)=1$ iff v is a nonpendant, (Each $mark(v)$ has the initial value 0). After that, sort $\{<mark(v),v>|1\leq v\leq m\}$ to seperate the pendants from the nonpendants. Finally, sort the pendants in ascending order by preorder by sorting the set $\{<pre(v),v>|mark(v)=0\}$. Let p be the number of pendants and $rank(v)$ be the position of v in the sorted sequence. Add (u,v) to A_2 if $rank(u)=rank(v)+\lfloor p/2\rfloor$. All these steps take at most $O(\lg m \lg\lg m)$ time with m processors. ■

With the help of Lemmas 2.22 and 2.23, we are ready to analyze the performance of Algorithm Brconnect.

Theorem 2.24: Algorithm Brconnect runs in $O(n/K+\lg^2 n)$ time with $nK(K\geq 1)$ processors on a PRAM.

Proof: In Step 1, the bridges and bridge-connected

components of $G(V,E)$ are determined. This takes $O(n/K+\lg^2 n)$ time with $nK(K\geq 1)$ processors[CHIN81,CHIN82]. In Step 2, to condense $G(V,E)$ into $G(V_0,E_0)$, we just have to determine V_0 and E_0. Clearly, V_0 can be formed by adding to it exactly one vertex from each bridge-connected component of G (each of these vertices serves to represents a bridge-connected component). For convenience, we choose the smallest-numbered vertex from each component. As a result, we immediately have $V_0=\{v|v=C(v)\}$ and $E_0=\{(C(u),C(v))|(u,v)$ is a bridge of $G\}$. Note that the array $\{C(v)|1\leq v\leq n\}$ and the bridges are determined in Step 1, and as a consequence, determining V_0 and E_0 takes $O(1)$ time with n processors. Steps 3,4 and 5 takes $O(n/K+\lg^2 n)$ time with $nK(K\geq 1)$ processors by Lemmas 2.22 and 2.23 (note that $m\leq n$). Finally, in Step 6, due to the way we construct V_0, the vertices of V_0 are also vertices of V, therefore, $A=A_1 \cup A_2$ and no transformation is required. Thus this step takes $O(1)$ time. Hence, the theorem follows.■

2.10 The Biconnectivity Problem

2.10.1 Introduction

Like the bridge-connectivity problem, this problem also consists of two subproblems, namely finding the set of all biconnected components and finding the set of all separation vertices in an undirected graph. The previous best results on this problem were due to Savage and Ja'Ja'[SAVA81]. They

presented two algorithms; one runs in $O(\lg^2 n)$ time with $n^3/\lg n$ processors while the other runs in $O(\lg^2 n \lg B)$ time with $|E|n+n^2\lg n$ processors where B is the number of biconnected components in the graph.

In this section, the algorithm we present could run in $O(\lg^2 n)$ time with only $n\lceil n/\lg^2 n\rceil$ processors.

2.10.2 Finding all Biconnected Components in an Undirected Graph

In this section, we present an optimal parallel algorithm for finding all biconnected components of a connected, undirected graph $G(V,E)$. Since a biconnected component can be completely determined by its vertex set, it suffices to find the vertex sets of all the biconnected components of G. Our algorithm is based on the following lemma.

Lemma 2.25: (i) For each edge $(a,b)\in E$ there exists a unique biconnected component in G containing the edge.

(ii) All edges in the same simple cycle in G belong to the same biconnected component in G.

(iii) Let C_1 and C_2 be two simple cycles having an edge in common. Then C_1 and C_2 belong to the same biconnected component in G.

The general strategy of our algorithm is as follows. Given the undirected graph G, we begin by constructing an

inverted spanning tree T of G. From T, we generate a set of fundamental cycles of G. From Lemma 2.25(i) and (ii), every fundamental cycle falls entirely within a unique biconnected component. We then use these fundamental cycles as the building blocks and begin to merge those cycles having common edges into bigger circuits. By Lemma 2.25(iii), each of these circuits belongs to exactly one biconnected component. We then merge the circuits having common edges into yet bigger circuits. This process is carried on until no further merge is possible. Then every circuit generated contributes to a biconnected component in G.

To make the fundamental cycles easier to handle, we remove the non-tree edge from each of them. This is legitimate because no two fundamental cycles can possibly intersect at a non-tree edge. The advantage is that the number of edges involved is now reduced from $O(n^2)$ to $n-1$. This modification also implies that we are in fact manipulating the branches of T rather than the fundamental cycles and that the process of merging the fundamental cycles has become the process of merging branches into subtrees. Consequently, when the merging process is complete, the result is a set of trees each of which is a spanning tree of a distinct biconnected component of G. Obviously, the vertex sets of these trees are the vertex sets of the biconnected components of G.

The merging process cannot be time-consuming for otherwise the performance of the entire algorithm will be

degraded. The method we use in our algorithm is to reduce the merging process to the problem of finding the connected component of an undirected graph derived from the inverted spanning tree T.

Definition: Let $T(V,E')$ be an inverted spanning tree of $G(V,E)$. Let $e_1=\langle a,\mathrm{F}(a)\rangle, e_2=\langle b,\mathrm{F}(b)\rangle \in E'$. Then

$e_1 \Delta e_2$

iff (i) e_2 is on $[a \ast\!\!\rightarrow \mathrm{HLCA}(a)]$ or e_1 is on $[b \ast\!\!\rightarrow \mathrm{HLCA}(b)]$;

or (ii) $(a,b) \in E-E'$ and neither $a \leq b$ nor $b \leq a$ in T.

From the definition, if $e_1 \Delta e_2$ then e_1 and e_2 belong to the same fundamental cycle. It is easily shown that if $e_1 \Delta e_2$ and $e_2 \Delta e_3$, then e_1 and e_3 belong to the same simple cycle in G. This is easily generalized to:

Lemma 2.26: If $e_1 \Delta e_2$, $e_2 \Delta e_3$, ..., $e_{t-1} \Delta e_t$, then there exists a simple cycle in G containing both e_1 and e_t.

Definition: Let $G(V,E)$ be an undirected graph and $T(V,E')$ be its inverted spanning tree. Then $G''(E',E'')$ is an undirected graph in which $(e_1,e_2) \in E''$ iff $e_1 \Delta e_2$.

The following theorem establishes the relationship between G and G''.

Theorem 2.27: e and e' belong to the same connected component in G'' iff e and e' belong to the same biconnected

component in G.

Proof: Let e and e' belong to the same connected component in G''. Then there exists a path : $e, e_1, \ldots, e_t, e'$ in G''. This implies that $e \Delta e_1$ and $e_1 \Delta e_2$ and ... and $e_t \Delta e'$. By Lemma 2.26, e, e' belong to the same cycle in G. By Lemma 2.25(ii), e and e' belong to the same biconnected component in G.

Let e and e' belong to the same biconnected component in G. Then there exists a simple cycle C containing e and e' in G. Let Γ be the set of fundamental cycles such that $C = \uplus \Gamma$. Construct an undirected graph $H(\Gamma, \Xi)$ such that $(C_1, C_2) \in \Xi$ iff C_1 and C_2 have a common edge. Clearly, H cannot be disconnected for otherwise C cannot be a simple cycle. Let $P = \{C_i\}_{i=1}^{t}$ be the shortest path in H such that $e \in C_1$, $e' \in C_t$. Let e_i be a common edge of C_i and C_{i+1}, $1 \leq i < t$.

Let (a_i, b_i) be the edge in $G-T$ determining C_i $1 \leq i \leq t$. Let $e(a_i)$, $e(b_i)$ be the edges in T such that $e(a_i) = \langle a_i, F(a_i) \rangle$ and $e(b_i) = \langle b_i, F(b_i) \rangle$; then in each C_i, we have: (i) $e(a_i) \Delta e(b_i)$ and ($e_{i-1} \Delta e(a_i)$ or $e_{i-1} \Delta e(b_i)$) and ($e_i \Delta e(a_i)$ or $e_i \Delta e(b_i)$); or (ii) $e_{i-1} \Delta e(a_i)$ and $e_i \Delta e(a_i)$; or (iii) $e_{i-1} \Delta e(b_i)$ and $e_i \Delta e(b_i)$. In any of the cases, there is a path from e_{i-1} to e_i in G''. In particular, there is a path from e to e_1 and a path from e_t to e' in G''. Joining all these paths together, we have a path from e to e' in G''. Hence, e and e' belong to the same connected component in G''. ∎

Algorithm:Biconnect

```
1. Find an inverted spanning tree $T(V,E')$ of $G(V,E)$;
2. Compute HLCA($v$) $\forall v \in V$;
3. Construct an undirected graph $G''(E',E'')$ such that
      $(e_1,e_2) \in E''$ iff $e_1 \Delta e_2$.
4. Find the connected components $\{B_j\}$ of $G''$. (* Note: Every
   connected components of $G''$ uniquely determines the
   vertex set of a biconnected component in $G$ and vice
   versa. *) ■
```

Theorem 2.28: Algorithm Biconnect runs in $O(n/\mathrm{K}+\lg^2 n)$ time with $n\mathrm{K}(\mathrm{K}\geq 1)$ processors on a PRAM.

Proof: With $n\mathrm{K}(\mathrm{K}\geq 1)$ processors available, Step 1 takes $O(n/\mathrm{K}+\lg^2 n)$ time (Theorem 2.5). Step 2 takes $O(n/\mathrm{K}+\lg n\cdot\lg\lg n)$ time (Theorem 2.17). Step 3 can be carried out as follows: Construct an adjacency matrix M'' for G'': for every $e \in E'$, $M''[e,e']$ and $M''[e',e]$ are set to 1 iff (i) e' is on the path $[a \ast\!\!\rightarrow \mathrm{HLCA}(a)]$ or (ii) (a,b) is in $G-T$ and neither $a \not\leq b$ nor $b \not\leq a$ in T, where $e=\langle a,\mathrm{F}(a)\rangle$ and $e'=\langle b,\mathrm{F}(b)\rangle$. Due to $|E'|=O(n)$ and the availablity of $\mathbf{F}^+$, testing the above conditions takes $O(n/\mathrm{K})$ time with $n\mathrm{K}(\mathrm{K}\geq 1)$ processors(Lemma 2.1). Step 4 takes $O(n/\mathrm{K}+\lg^2 n)$ time[CHIN81,CHIN82]. Hence, Algorithm Biconnect takes $O(n/\mathrm{K}+\lg^2 n)$ time with $n\mathrm{K}(\mathrm{K}\geq 1)$ processors. ■

For completeness, we would like to point out that the algorithm for finding all biconnected components can be used to determine the set of all bridges as well. This is based on the fact that an edge e of G is a bridge iff e is a biconnected component of G.

2.10.3 Finding all the Separation Vertices in an Undirected Graph

Let $T(V,E')$ be an inverted spanning tree of $G(V,E)$ and B_j is a biconnected component of G. Then $B_j \wedge T$ must be connected and is thus a tree. Let $a \in V$. If a is not the root r of T, then a is a separation vertex of G iff a is the root of $B_j \wedge T$ for some biconnected component B_j of G. Moreover, r is a separation vertex iff r is the root of $B_i \wedge T$ and $B_j \wedge T$, where B_i, B_j are two distinct biconnected components of G. These ideas are embodied in the following lemma.

Lemma 2.29: Let $T(V,E')$ be an inverted spanning tree of $G(V,E)$; r be the root of T and $\{B_k\}_{k=1}^{t}$ be the set of biconnected components of G.

a is a separation vertex of G

iff a is the root of $B_j \wedge T$ for some j, if $a \neq r$;

or a is the root of $B_i \wedge T$ and $B_j \wedge T$ for some $i \neq j$, if $a = r$.

Proof: Only if part: Let a be a separation vertex of G. There exist biconnected components B_i, B_j, $i \neq j$ such that a belongs to both B_i and B_j([AHO74],Lemma 5.4,p.181).

If $a \neq r$, we may assume without loss of generality that $\langle a, F(a)\rangle$ belongs to $B_i \wedge T$. Let r_j be the root of $B_j \wedge T$. There exists a path P_1 in B_j from a to r_j. There also exist a path P_2 in T from r_j to $LCA(a,r_j)$ and a path P_3 in T from a to $LCA(a,r_j)$. Clearly, P_2 and P_3 contain no edges in B_j. But then P_1, P_2 and P_3 give rise to a simple cycle in G which will contradict the fact that B_i, B_j are biconnected

components unless $a=r_j$. Thus a is the root of $B_j \wedge T$.

If $a=r$, then there exists a path P from r_i to a in T. Since $B_i \wedge T$ is connected, all the edges on P must belong to $B_i \wedge T$. But then r_i cannot be the root of $B_i \wedge T$ unless $a=r_i$. The same argument implies that $a=r_j$.

If part: Let $a=r$ and a is the root of $B_i \wedge T$ and $B_j \wedge T$ where $i \neq j$. Let s_i and s_j be a son of a in $B_i \wedge T$ and $B_j \wedge T$ respectively. Suppose after removing a from G, the resulting graph remains connected. Then there must be a path from r_i to r_j in G not passing through a. However, this path and the edges (a,r_i), (a,r_j) will form a cycle in G which implies that G_i and G_j cannot be biconnected components. Therefore the removal of a from G must disconnect G which means that a is a separation vertex.

Let $a \neq r$ and a is the root of some $B_j \wedge T$. consider F(a) and s_j where s_j is a son of a in $B_j \wedge T$. F(a) does not belong to B_j for otherwise a cannot be the root of $B_j \wedge T$. By applying an argument similar to the one above, we can show that removing a from G would result in disconnecting F(a) and s_j. Hence a is a separation vertex of G.■

As a consequence of Lemma 2.29, the algorithm for finding the biconnected components can be used to determine the set of all separation vertices of G as follows.

Theorem 2.30: The set of separation vertices can be found in $O(n/\mathrm{K}+\lg^2 n)$ time with $n\mathrm{K}(\mathrm{K} \geq 1)$ processors on a PRAM.

Proof: First, the set of all biconnected components is determined. This takes $O(n/\mathrm{K}+\lg^2 n)$ time with $n\mathrm{K}(\mathrm{K}\geq 1)$ processors(Theorem 2.28). Next, the head of each $e\epsilon E'$, head(e), is determined. This obviously takes $O(1)$ time with $n\mathrm{K}$ processors. Then the set of all head(e)'s are divided into groups such that those e's belonging to the same biconnected component have their head(e)'s grouping together. This involves sorting and takes $O(\lg n\cdot\lg\lg n)$ time with n processors or $O(\lg n)$ time with $n\lg n$ processors[BORO82]. Finally, the head(e) with the smallest depth in each group is selected, these head(e)'s form the set of separation vertices. r is included in the set iff r is selected from two or more groups. This step takes $O(n/\mathrm{K}+\lg\mathrm{K})$ time with $n\mathrm{K}$ processors(Lemma 2.2). ■

Finally, to determine the biconnectivity of a connected, undirected graph G. We can check the numbers of separation vertices it has. Clearly, G is biconnected iff there is no separation vertices. This takes $O(n/\mathrm{K}+\lg^2 n)$ time with $n\mathrm{K}(\mathrm{K}\geq 1)$ processors.

2.11 Conclusions

In the preceding sections, we assume in most cases that $n\mathrm{K}$, the number of processors available, satisfies the condition $\mathrm{K}\geq 1$. This means that the number of processors available is not less than n. In fact, this assumption is made for convenience only because most of the previous work assumed unbounded parallelism. To make use of some of those

results in the course of developing our algorithms, we found it most convenience to assume $K \geq 1$. Nevertheless, it is not difficult to extend our results to cases where $0<K<1$ if Brent's theorem is used.

Theorem 2.31:[BREN74] If a synchronized computation C consisting of a total of q operations can be performed in t parallel time units with sufficiently many processors, then C can be performed in $\lceil (q-t)/p \rceil + t$ time units with $p(>0)$ processors.

Using the above theorem, it is easily shown that Lemma 2.2 can be generalized to: "Given an array of n^2 elements, $\{a_{ij}\}$ $1 \leq i,j \leq n$, and $nK(K>0)$ processors, $A(i)=a_{i1}*a_{i2}*...*a_{in}$, $1 \leq i \leq n$ can be computed in $\lceil (n^2-n-\lg n)/nK \rceil + \lg n = O(n/K+\lg n)$ time units." Similarly, Preparata's sorting algorithm can be executed in $\lceil n\lg^2 n/nK \rceil + \lg n = O(\lg^2 n/K+\lg n)$ time units if $nK(K>0)$ processors are available.

Extension of all of our results from $nK(K \geq 1)$ to $nK(K>0)$ can be accomplished in a similar way.

The parallel algorithms presented in this chapter are optimal for dense graphs except for the problem of finding the lowest common ancestors of vertex pairs in a directed tree, and the problem of finding all fundamental cycles in an undirected graph. If an optimal algorithm for finding the lowest common ancestors running in $O((n+q)/nK)$ time with

nK(K>0) processors is found, then the performance of the algorithm for finding the fundamental cycles is also improved without any modification. Moreover, this achievement will provide us with an alternate way to compute HLCA(v),$\forall v \in V$, which is crucial in the design of optimal parallel algorithms for the last four problems.

We feel that several techniques we use in this chapter deserve further attention as they may be useful in developing efficient algorithm for other graph theoretic problems or even for problems in other disciplines.

The first is the one used in handling graph theoretic problems which are strongly related to cycles. If we were to handle all the cycles directly, we could hardly expect the resulting algorithm to be polynomial with respect to the time-processor product because the number of cycles in a graph can be exponentially large. The technique we use is to restrict our domain of consideration from the set of cycles to the set of fundamental cycles (note that there are at most $O(n^2)$ of them). We also reduce the number of egdes to be considered from $|E|$ to n-1 by constructing an inverted spanning tree T for the given graph G and considering only the edges in T. This elaboration allows us to start with a managable number of items which require no more than $O(n^4)$ operations. Then by computing the function HLCA(u)'s, much of the information conveyed by the fundamental cycles can be stored under the HLCA(u)'s. Consequently, the possible number of operations is further reduced to $O(n^2)$ which makes

the $O(n^2)$ time-processor product possible. We believe that this technique may prove to be useful in other graph theoretic problems which are cylce-oriented.

The second is described in Lemma 2.2 which simply says that to compute an associative operation involving n items, if the well-known recursive-doubling technique is used, we need $n/2$ processors to achieve the $O(\lg n)$ time bound. However, if we have only $n/\lg n$ processors available, then we can still achieve the $O(\lg n)$ time bound with only a slightly larger constant factor. This technique is very useful as it allows us to reduce the number of processors used without affecting the order of magnitude of time. The technique was known previously but was not properly utilized.

The third one makes use of the observation that if a computation requires a number of iterations and after each iteration, the problem size is reduced by at least half, then the total amount of time required (in terms of order of magnitude) to complete the computation is the same as that required by the first iteration. Specifically, $\Sigma_{i=0}^{k} T/2^i < 2T$ for any $k > 0$.

Chapter 3

IMPLEMENTATION ON THE MMM MODEL

3.1 Introduction

In this chapter, we propose a general computer model, called *MMM*, which includes all the parallel computer models on which an ordinary matrix multiplication algorithm exists. Since almost every existing computer model has an algorithm for the matrix multiplication problem, the model proposed has a great degree of generality. In fact, it includes all of the well-known existing parallel computer models listed below:

MCN(Mesh connected Networks)[CANN69,DEKE81,ATAL82],
PSN(Perfect Shuffle Networks)[STON71],
CCC(Cube-connected Cycles)[PREP81],
OTN(Orthogonal Tree Networks)[NATH81],
OTC(Orthogonal Tree Cycles)[NATH81],
SIMD-CCC(SIMD Cube-connected Computers)[DEKE81],
PRAM(SIMD Shared Memory Model allowing read conflicts)[WYLL79]
WRAM(SIMD Shared Memory Model allowing read and write conflicts)[SHIL81].

Let $O(t(n))$ and $H(n)$ denote the time and hardware resources(in terms of number of processors and chip area) required by the nxn ordinary matrix multiplication algorithm. We shall show that our algorithms take *at worst* a

factor of $\max(\lg d,\lg d'')+1$, $1\le d,d''\le n$, more time and the same amount of hardware resources as those required by the matrix multiplication algorithm on the *MMM*. Since on many of the well-known existing models, the matrix multiplication algorithm takes *at most* $O(\lg n)$ time and $H(n)$ hardware resources, our algorithms are therefore *bounded above*[8] by $O(\lg n*(\max(\lg d,\lg d'')+1))$ in time and $H(n)$ in hardware resources on those models. This result turns out to be very efficient as it outperforms the previously known best algorithms on many models.

3.2 The Computer Model MMM

3.2.1 Definitions

definition: Let (S,+,*,0,1) be a ring and M_n be the set of nxn matrices over S. An **ordinary matrix multiplication algorithm** for M_n is an algorithm which takes advantage of *only* the associative property of + in multiplying any two matrices in M_n.

Note that the well-known Strassen algorithm[STRA69] is not an ordinary matrix multiplication algorithm because it makes use of the additive inverse property of +. There are two reasons why we consider only ordinary matrix

---

[8] The term 'bounded above' need some clarification. Here we mean that the algorithms will take $O(t(n)*(\max(\lg d,\lg d'')+1))$ time and $H(n)$ hardware resources if the algorithms are indeed implemented in a way using matrix multiplication. However, as it will be clear in the following section that our algorithms do not rely on matrix multiplication, other more efficient techniques could be used if they were available.

multiplication algorithms here. The first is because for most of the existing computer models, the only known algorithm for multiplying matrices is an ordinary matrix multiplication algorithm. The second reason is that in the rest of this chapter, we will frequently encounter matrices whose elements are chosen from closed semirings[AHO74], and closed semirings do not possess the additive inverse property. As a result, matrix multiplication algorithms for matrices over a ring which make use of the additive inverse property cannot be applied to these matrices.

The *MMM*(**Matrix Multiplication Model**) has the following features:

(i) there exists an ordinary matrix multiplication algorithm;

(ii) each processor contains a constant number of registers and is capable of carrying out any of the operations +, −, *[9], $\vee$, $\wedge$, $\neg$, =, ≠, ≤, ≥ in constant time;

(iii) communication between interconnected processors and between registers within the same processor takes constant time.

In representing the given undirected graph, an adjacency matrix M is used. The entry $M[i,j]$ of M is stored in the M register of processor $PE[i,j]$, $1 \le i,j \le n$. In general, register, say A, in $PE[i,j]$ is denoted by $A[i,j]$. Again, without loss of generality, we assume G is connected and the

---

[9] As a matter of fact, multiplication is not used in our algorithms, the *'s appearing in the algorithms are just a shorthand of the **if...then...else** statement.

vertex set $V=\{1,2,3,\dots,n\}$, throughout this chapter. We use d to denote the diameter of G, Σ to denote the summation of integers and an APL type of syntax to describe our algorithms. As a result, 0 will represent both the integer zero and the boolean constant 'false' and 1 will represent both the integer one and the boolean constant 'true'. As an example, $c*(a=b)$ is equivalent to **if** $a=b$ **then** c **else** 0.

Definition: A function f is called an **extended monadic function w.r.t. i, j** if the arguments of f are of the form $OP[i,j]$ where OP is either the name of a register or a function of i, j. We denote it by $f[i,j]$.

The following lemma dominates the rest of this chapter.

Lemma 3.1: The following operation could be carried out on the *MMM* using the same order of magnitude of time and hardware resources as the ordinary matrix multiplication algorithm.

$M[i,j]:=f_3(\Xi_{k=1}^{n}(\Pi(f_1[i,k],f_2[k,j])))\ \forall\ i,j,\ 1\le i,j\le n$, where $M[i,j]$ is a register of processor $PE[i,j]$. f_1, f_2, f_3 are extended monadic functions w.r.t. i,k; k,j and i,j respectively. Π is a composite function of the arithmetic and boolean operations mentioned in the definition of *MMM* and Ξ is an associative operator.

Proof: Trivial. ■

3.2.2 Some Perliminary Results

In the following sections, in proving the resource complexities of each step of the algorithms, we shall use the following strategy: we show that the step can be carried out by a method similar to that used by the matrix multiplication algorithm. The advantage of this strategy should be obvious as it allows us to carry out our analysis without having to deal with the detailed structure of the model (e.g. how the processors are connected together). A typical example is data routing which is always needed in parallel algorithms and whose implementation and efficiency are greatly model-dependent. Using the above strategy, data routing can be handled in a model-independent way.

Broadcasting the contents of a register columnwise or rowwise is needed frequently in subsequent discussions. We give a bound on its resource complexities in the following lemma using the above-mentioned strategy.

Lemma 3.2: Let $PE[i,j]$ $1 \le i,j \le n$, be a set of processors. The time and hardware resources needed to broadcast the contents of register $M[a,b]$ columnwise (rowwise) is at worst the same as that needed by the ordinary matrix multiplication algorithm on the *MMM*.

Proof: To broadcast the contents of $M[a,b]$ columnwise, we perform

$$M[w,b] := \Sigma_k((dummy[w,k]*0)+(M[k,b]*(k=a))) \quad 1 \le w \le n;$$

or simply,

$M[w,b] := \Sigma_k((dummy[w,k]*0)+M[k,b])$, $1 \le w \le n$, if $M[a,b]$ is the only possible non-zero term in the column.

Here *dummy* can be any register as its appearance is just to ensure that the resulting expression conforms to the one stated in Lemma 3.1, it is irrelevant to the computation. Clearly, $M[w,b]=M[a,b]$, $1 \le w \le n$. By Lemma 3.1, the lemma follows.

Broadcasting rowwise can be handled in the similar way. ■

We have to emphasize that we do not mean that broadcasting the contents of a register columnwise or rowwise has to be actually done in the above way. We merely want to show that its complexity is bounded above by that of matrix multiplication.

The following are some basic results which will be referred to frequently in the rest of this chapter.

Lemma 3.3: The following operations can be carried out in $O(t(n))$ time with $H(n)$ hardware resources on the *MMM*.

(i) $\mathcal{R}[u,v] := \Xi_{k=1}^{n} f(u,k)$, $1 \le u,v \le n$;

(ii) $\mathcal{R}[u,j] := \Sigma_{k=1}^{j} f(u,k)$, $1 \le j \le n$, $1 \le u \le n$.

where $\mathcal{R}$ is any register, $f(u,k)$ is an extended monadic function w.r.t. u, k and Ξ is an associative operator.

Proof:

$\Xi_{k=1}^{n} f(u,k)$ is equivalent to $\Xi_k(f(u,k)+(dummy[k,v]*0))$.

Similarly,

$\Sigma_{k=1}^{j} f(u,k)$ is equivalent to $\Sigma_k(f(u,k)*(k \le j))$.

From Lemma 3.1, the lemma follows. ■

Lemma 3.4: Let $O(t'(n))$ and $H'(n)$ be the time and hardware resources required by the all-pair shortest path algorithm on the *MMM* and G be an unweighted (directed or undirected) graph with diameter d; then

$$t'(n)=t(n)*(\lg d+1) \text{ and } H'(n)=H(n)$$

Proof: Let M be an adjacency matrix of G.

Construct matrix D such that

$$D[i,j] = \begin{cases} 1 \text{ if } M[i,j]=1 \text{ and } i\neq j; \\ 0 \text{ if } i=j; \\ +\infty \text{ if } M[i,j]=0. \end{cases}$$

Compute the matrix D^d as follows:

$D^1 = D$

$D^{2**i}[u,v]=\mathbf{min}_k(D^{2**(i-1)}[u,k] + D^{2**(i-1)}[k,v]),\ i\geq 1.$

A simple induction will reveal that $D^{2**i}[u,v]$ contains the length of the shortest path from u to v consisting of no more than 2^i edges. Therefore after $\lg d$ iterations, $D^d[u,v]$ will contain the shortest distance from u to v in G. One more iteration is required to verify that D^d has been computed. By Lemma 3.1, $t'(n)=t(n)*(\lg d+1)$ and $H'(n)=H(n)$. ■

Lemma 3.5: Computing the transitive closure of an adjacency matrix can be done in $t''(n)$ time with $H''(n)$ hardware resources where $t''(n)\leq t'(n)$ and $H''(n)\leq H'(n)$.

Proof: Let the transitive closure matrix be M^*; then $M^*[a,b]=1$ iff $D^d[a,b]\neq+\infty$. ■

3.3 Constructing a Breadth-first Search (Directed) Spanning Forest of an Undirected Graph

In this section, we shall present an efficient algorithm for constructing a directed breadth-first search(BFS) spanning forest for an undirected graph on the *MMM*. The method used was first 'implicitly' given by Savage for the PRAM[SAVA77]. It also appeared in [DEKE81] and [ATAL82].

Theorem 3.6: Given an nxn adjacency matrix M of an undirected graph $G(V,E)$, a directed BFS spanning forest for G can be determined in $O(t(n)\cdot(\lg d+1))$ time with $H(n)$ hardware resources on the *MMM*.

Proof: We shall construct an adjacency matrix T for the BFS spanning forest (since an inverted spanning tree is more convenient in some cases, the transpose of T, T' is also constructed).

First, compute the all-pair shortest path matrix D^d for G and the transitive closure matrix M^* using Lemmas 3.5 and 3.6. Then for each connected component of G, choose the smallest-numbered vertex in it as the root of its spanning tree. Since every smallest-numbered vertex of a connected component satisfies the following property, namely, v is the smallest-numbered vertex in a connected component iff $M^*[v,k]=0$, $\forall k<v$, the set of all these vertices can be determined easily as follows: compute the partial sums $Rank[u,j]:=\Sigma_{k=1}^{j} M^*[u,k]$, $1\le j\le n$, (Lemma 3.3(i)); then every

processor computes locally $Rep[u,j]:=(Rank[u,j]=1)\wedge(u=j)$, $1\leq u,j\leq n$. After this step, it should be clear that $Rep[r,r]=1$ iff r is the smallest-numbered vertex of a connected component iff r is the root of a spanning tree (note that Rep is a boolean array).

After all the roots r of the BFS spanning forest are determined, the level of every vertex in the forest is also determined. This is because $level(v)=D^d[r,v]+1$, $\forall v\in V$, where r is the root of the tree containing v. We shall store $level(v)$ into $level[v,v]$. This is accomplished as follows:

$level[r,v] := D^d[r,v]+1$;

(Broadcast columnwise) $level[k,v]:= level[r,v]\ \forall v,k\in V$;

At this point, $level[v,v]=level(v), \forall v\in V$.

Next, select a father for each vertex v which is not a root. This is accomplished in two steps. In the first step, all the vertices whose levels are one less than that of v are identified:

(Broadcast rowwise) $level[v,k] := level[v,v]\ \forall v,k\in V$;

$F'[v,j]:=\vee_k(level[v,k]=((1+level[k,j])*(k=j)))$.

The second statement needs some explanation: after broadcasting $level[v,v]$, $\forall v\in V$ rowwise, $level[v,w]=level(v)$, $\forall v,w\in V$. As a result, the right-hand side of the statement is equivalent to $\vee_k(level(v)=$ **if**$(k=j)$**then**$(1+level(k))$**else**$0)$ which in turns is equivalent to **if** $(level(v)=(1+level(j)))$ **then** 1 **else** 0. Hence, $F'[v,j]=1$ iff $level(v)=level(j)+1$.

In the second step, the largest-numbered vertex which is one level higher than v and is adjacent to v in G is selected as

the father of v in the BFS spanning forest:

$F[v,v]:=\max_k(\text{if } (F'[v,k]\wedge M[v,k]) \text{ then } k \text{ else } 0;$

(Lemma 3.3(i))

Note that $F[v,v]$, for $v\neq r$, contains the father $F(v)$ of v in the BFS spanning forest.

Finally, construct an adjacency matrix T and its transpose T' to represent the BFS spanning forest. This is accomplished by the following computations:

(Broadcast columnwise:) $F'[k,v] := F[v,v]\ \forall v,k\epsilon V;$

(Broadcast rowwise:) $F[v,k] := F[v,v]\ \forall v,k\epsilon V;$

$T[w,v] := (w=F'[w,v])\ \forall v,w\epsilon V;$

$T'[v,w] := (w=F[v,w])\ \forall v,w\epsilon V.$

Thus, T and T' are boolean matrices such that $T[u,v]=1$ (resp. $T'[u,v]=1$) iff u is the father (resp. a son) of v.

From Lemmas 3.1, 3.2, 3.4 and 3.5, we have: finding a directed BFS spanning forest of an undirected graph takes $O(t(n)\cdot(\lg d+1))$ time with $H(n)$ hardware resources. ■

3.4 Finding the Lowest Common Ancestors of all Vertex Pairs in a Directed Tree

In this section, we implement the algorithm for finding the lowest common ancestors presented in Chapter 2 on the *MMM*.

Theorem 3.7: Given an adjacency matrix T of a directed tree with diameter d, computing the lowest common ancestors of all vertex pairs of the directed tree can be done in

$O(t(n)\cdot(\lg d+1))$ time with $H(n)$ hardware resources on the *MMM*.

Proof: First, construct the transpose T' of T as follows: every processor executes the statement $F[u,v]$:= **if** $T[u,v]$ **then** u **else** 0; locally. Since there is one and only one nonzero $F[u,v]$ value in each column, we may use Lemma 3.2 to broadcast these nonzero $F[u,v]$'s columnwise. After this step, $F[v,v]$ contains the father of v in the directed tree. Then perform:

(Broadcast rowwise:) $F[v,k]:=F[v,v],\ \forall v,k\in V;$

$T'[v,u]:=u=F[v,u],\ \forall u,v\in V.$

From Lemma 3.2, this step takes $O(t(n))$ time with $H(n)$ hardware resources.

Next, compute the transitive closure T^* and $(T')^*$ of T and T' respectively. By Lemma 3.6, this step takes $O(t(n)\cdot(\lg d+1))$ time with $H(n)$ hardware resources. Note that in the course of computing the transitive closures, the *level* of each vertex is also determined (recall that $level(v)=1+D^d[r,v]\ \forall v\in V$, where r is the root of T) and $level(v)$ is stored in $level[v,v]$.

Finally, compute the matrix LCA:

(Broadcast rowwise:) $level[v,w]:=level[v,v];$

LCA$[i,j]$:= $(\max\leq)_k\{(T')^*[i,k]*(T^*[k,j]*k)\}$.

The above expression in the braces should be interpreted as:

```
if k is an ancestor of i
then if k is an ancestor of j then k
                                else 0
```

else 0;

The evaluation of the binary operation $(\max\leq)\{a,b\}$ needs some explanation. We proceed in two time units. In the first time unit, a and b are transferred simultaneously from processors $PE[i,k]$ and $PE[k,j]$ respectively to a processor $PE[i,j,k]$ at which the binary operation is to be carried out. In the second time unit, $level(a)$ and $level(b)$ are transferred simultaneously from processors $PE[i,k]$ and $PE[k,j]$ to $PE[i,j,k]$. The values of $level(a)$ and $level(b)$ are then compared in that processor and if $level(a)$ is greater, then a is the value of $(\max\leq)\{a,b\}$, otherwise b is the value.

Computing the matrix LCA takes $O(t(n))$ time with $H(n)$ hardware resources.■

3.5 Finding a set of Fundamental Cycles of an Undirected Graph

As with Section 5 of Chapter 2, we shall construct the matrices $\mathbf{F}^+$, LCA and P^+ to represent the fundamental cycles on the *MMM*. Since LCA has been discussed in the last section and P^+ can be easily determined from *level*: $(P^+(v)=)P^+[v,v]:=n-level[v,v]$, we shall discuss only the construction of $\mathbf{F}^+$.

Assuming that an adjacency matrix M of $G(V,E)$ is given. We construct the matrices T and T' for a BFS spanning tree of G. Clearly, the diameter of the BFS spanning tree is not greater than that of G. Then using Theorem 3.4, we compute

the all-pair shortest path matrix $(T')^d$ for T'. Since for any vertex u in an inverted tree, all vertices reachable from u are located on the path from u to the root, therefore, the uth row of $(T')^d$ contains exactly one i for each i in the range $[0,level(u))$, and contains no j in the range $[level(u),n]$. Consequently, $F^k(u)$ can be computed as follows:

$F^+[u,k]:=\Sigma_j((T')^d[u,j]\simeq(j,k))$,

where $\simeq$ is defined as $a\simeq(b,c) \equiv$ **if** $a=c$ **then** b **else** 0. Computing $\simeq$ can be done in a manner similar to that used for computing $(\max\lesssim)\{a,b\}$. Specifically, we proceed in two time units. In the first time unit, a and b are transferred simultaneously from processors $PE[u,j]$ and $PE[j,k]$ respectively to a processor $PE[u,k,j]$. In the second time unit, c is transferred from $PE[j,k]$ to $PE[u,k,j]$. a and c are then compared in that processor. If they are equal, b is the value of the computation, otherwise the result is 0.

Finally, adjusting the array $\mathbf{F}^+$ is straightforward and takes no more than $O(t(n)\cdot(\lg d+1))$ time with $H(n)$ hardware resources.■

3.6 2-coloring an Undirected Graph

We shall implement Algorithm Bipartite on the *MMM* in this section.

Theorem 3.8: Given an adjacency matrix M of an undirected graph G, the 2-colorability problem can be solved in

$O(t(n)\cdot(\lg d+1))$ time with $H(n)$ hardware resources.

Proof: We generate the matrices T and T' for a BFS spanning tree T of G using Theorem 3.6 and the adjacency matrix M' of G-T by computing $M'[i,j]:=M[i,j]\wedge\neg(T[i,j]\vee T'[i,j])$ locally at every processor. Note that M' is a boolean matrix such that $M'[i,j]:=1$ iff (i,j) is an edge in G but not in T. We then examine every fundamental cycle in G by testing if $level(i)\neq level(j)$, for every (i,j) in G-T, as follows:

(Broadcast rowwise:) $level[v,w]:=level[v,v]$;

(broadcast columnwise:) $level'[w,v]:=level[v,v]$;

$Flag[i,j]:=level[i,j]\neq level'[i,j]$.

The previous statement is equivalent to $Flag[i,j]:=level(i)\neq level(j)$. Here we employ a property of the BFS spanning trees: if (i,j) is an edge in G-T, then the difference between the levels of i and j cannot be greater than 1. As a consequence, the condition "$level[i,j]\neq level[i,j]$" is equivalent to that tested in Step 2(ii) of Algorithm Bipartite.

Now, we assume $Flag[i,j]$ has the initial value 1. We proceed to compute $\wedge_{i,j}Flag[i,j]$ as follows:

$Bipartite[i,j]:=\wedge_k Flag[i,k],\ \forall i,j\in V$; (Lemma 3.3(i));

$Bipartite[i,j]:=\wedge_k Bipartite[k,j],\ \forall i,j\in V$; (Lemma 3.3(i)).

At this point, G is 2-colorable(bipartite) iff $Bipartite[1,1]=1$.

Finally, if $Bipartite[1,1]=1$, we compute $Partition[v,v]:=Bipartite[v,v]\wedge(level[v,v]$ is odd) locally

at every processor. By Lemmas 3.1, 3.2, 3.5 and Theorem 3.6, the total time taken is $O(t(n)\cdot(\lg d+1))$ and the hardware resources needed are $H(n)$.■

3.7 The Bridge-connectivity Problem

In this section, we shall implement Algorithm Bridges presented in Chapter 2 on the *MMM*. We shall determine the set of bridge-connected components at the same time.

Theorem 3.9: Given the $n\mathrm{x}n$ adjacency matrix M of $G(V,E)$, the set of all bridges and bridge-connected components in G can be determined in $O(t(n)\cdot(\lg d+1))$ time with $H(n)$ hardware resources on the *MMM*.

Proof: We proceed in 6 steps.

In Step 1, we construct the matrices T and T' for a BFS spanning tree T of $G(V,E)$. As a consequence, the matrix *level* is also available. Recall that $level(v)=level[v,v]$, $\forall v \in V$, (Lemma 3.6).

In Step 2, we compute ℓLCA(i,j) which is the level of LCA(i,j) as follows:

Compute the transitive closures T^* and $(T')^*$ of T and
    T' respectively.

(Broadcast rowwise) $level[v,w] := level[v,v]$, $\forall v,w \in V$;

ℓLCA$[i,j]:=\max_k\{(T')^*[i,k]*(T^*[k,j]*level[k,j])\}$.

The expression in the above statement can be interpreted as **if** $(T')^*[i,k] \wedge T'^*[k,j]$) **then** $level(k)$ **else** 0. Note that in particular, ℓLCA$[v,v]=level(v)$ $\forall v \in V$.

In Step 3, we construct an adjacency matrix M' for G-T as well as the matrix ℓHLCA:

$M'[i,j] := M[i,j] \wedge (\neg(T[i,j] \vee T'[i,j]))$;

(note that $M'[v,v]=1\ \forall v \in V$)

$\ell\mathrm{HLCA}[v,v] := \min_k\{$**if** $M'[v,k]$ **then** $\ell\mathrm{LCA}[v,k]$ **else** $0\}$,

(Lemma 3.3(i));

note that $\ell\mathrm{HLCA}[v,v]$ contains the level of HLCA(v) and $0<\ell\mathrm{HLCA}[v,v]\leq level(v)\ \forall v \in V$.

In Step 4, we compute the array α:

(Broadcast rowwise:) $\ell\mathrm{HLCA}[v,w] := \ell\mathrm{HLCA}[v,v]\ \forall w \in V$;

$\alpha[v,w] := \min_k\{T^*[v,k]*\ell\mathrm{HLCA}[k,w]\}$;

Note that $\alpha[v,v]=\alpha(v)$. Moreover, $\alpha[v,w]=\alpha[v,v]\ \forall v,w \in V$.

In Step 5, we compute the matrix *Bridge*:

$Bridge[u,v] := \vee_k (T[u,k] \wedge ((\alpha[k,v] \geq level[k,v]) \wedge (k=v)))$;

The right-hand side of the above statement is equivalent to $T[u,v] \wedge (\alpha(v) \geq level(v))$, thus, $Bridge[u,v]=1$ iff (u,v) is a bridge in G.

Finally, in Step 6, we compute the bridge-connected components:

first remove the bridges:

$M''[i,j] := M[i,j] \wedge (\neg Bridge[i,j])$;

then compute the transitive closure $(M'')^*$.

From Theorems 3.5, 3.6 and Lemmas 3.1, 3.2, 3.3 and the fact that the diameter of M'' cannot be greater than d, we have: the bridge-connectivity problem can be solved in $O(t(n)*(\lg d+1))$ time with $H(n)$ hardware resources on the *MMM*.■

3.8 The Bridge-connectivity Augmentation Problem

In this section, we shall implement Algorithm Brconnect of Chapter 2 on the *MMM*. We proceed in a step-wise manner. First we show how G can be condensed into G_0 on the *MMM* in Lemma 3.10. Then we construct the edge set A_1 in Lemma 3.11. After that we discuss how a directed tree can be labelled in perorder on the *MMM* in Lemma 3.12 and how the edge set A_2 can be constructed in Lemma 3.13. Finally, in Theorem 3.14, we combine Lemmas 3.10-3.13 to derive the resource complexities of Algorithm Brconnect on the *MMM*.

Lemma 3.10: Given an adjacency matrix M of $G(V,E)$, the forest $G_0(V_0,E_0)$ can be constructed in $O(t(n)*(\lg d+1))$ time with $H(n)$ hardware resources on the *MMM*.

Proof: We shall construct an adjacency matrix M_0 to represent G_0.

First note that we can construct V_0 by picking a representative from each bridge-connected component of G. For convenience, we pick the smallest-numbered vertex from each bridge-connected component since this vertex can be determined easily by using the method described in Theorem 3.6. As a result, we have the matrix B-rep such $B\text{-rep}[v,v]=1$ iff v is the smallest-numbered vertex of a bridge-connected component iff $v\epsilon V_0$. Note that in the course of computing B-rep, we also compute the matrices *Bridges* and $(M'')^*$ (see Algorithm Bridges Steps 5 and 6).

To determine the edge set E_0, we first determine, for

each $u \epsilon V_0$, the set $V(u)=\{k|(j,k)$ is a bridge and u and j belong to the same bridge-connected component$\}$. Specifically, we compute:

$$\text{Cross-bridge}[u,k]:=\bigvee_j((M'')^*[u,j] \wedge \mathit{Bridge}[j,k]).$$

Note that for every $u \epsilon V_0$, Cross-bridge$[u,k]=1$ iff $k \epsilon V(u)$. Next, we replace each k in $V(u)$ with the vertex v in V_0 such that v represents the bridge-connected component containing k. This can be easily accomplished by computing:

$$\text{(Broadcast rowwise:) } B\text{-rep}[v,w]:=B\text{-rep}[v,v];$$

$$\text{(Broadcast columnwise:) } B\text{-rep}'[w,v]:=B\text{-rep}[v,v];$$

$$T_0[u,v]:=\bigvee_k \big((B\text{-rep}[u,k] \wedge \text{Cross-bridge}[u,k]) \wedge ((M'')^*[k,v] \wedge B\text{-rep}'[k,v]) \big).$$

The above statement should be interpreted as $T_0[u,v]:=(\exists k)(u \epsilon V_0$, and u crosses a bridge to reach k, where k is in the bridge-connected component as v where $v \epsilon V_0)$.

An adjacency matrix T_0 for G_0 is thus constructed.

From Lemmas 3.1, 3.2, and Theorem 3.9, we have the indicated time and hardware resource complexities. ■

Lemma 3.11 Given an adjacency matrix T_0 of $G_0(V_0,E_0)$, constructing the edge set A_1 takes $O(t(n)\cdot(\lg d+1))$ time with $H(n)$ hardware resources on the *MMM* where d is the diameter of G_0.

Proof: First compute T_0^* and *Rep*(Theorem 3.6) and then proceed in three steps.

In Step 1, we find the isolated vertices and select two pendants from each tree in G_0. These are the vertices having

degree ≤ 1. Therefore we begin by computing the degree of each vertex u:

$degree[u,u]:=\Sigma_{k=1}^{n}T_0[u,k]\ \forall u\epsilon V$, (Lemma 3.3(i));

(Note: $degree[u,u]$ is the degree of u in G).

Based on $degree$, we compute $Pen\text{-}iso[u,u]:=(degree[u,u]\leq 1)$ locally at every processor. Then $Pen\text{-}iso[u,u]=1$ iff u is a pendant or an isolated vertex of G_0. We assume $Pen\text{-}iso[u,v]=0$ for $u\neq v$.

The remaining part of Step 1 is devoted to isolating two pendants from each non-trivial tree of G_0 and the isolated vertices in G_0. The isolated vertices are labelled with -1 while the two pendants from the same tree are labelled with 1 and 2 respectively. The remaining vertices are labelled with 0. We begin with computing a boolean matrix Pi such that $Pi[u,v]=1$ iff v is a pendant of the tree represented by u in G_0 or v is an isolated vertex. Note that in the latter case u must equal to v.

(Broadcast columnwise:) $Pen\text{-}iso'[w,u]:=Pen\text{-}iso[u,u]$;

(Broadcast rowwise:) $Rep[u,w]:=Rep[u,u]$;

$Pi[u,v]:=Rep[u,v]\wedge T_0^*[u,v]\wedge Pen\text{-}iso'[u,v]$.

Using Pi, we rank the pendants of every non-trivial tree:

$PRS[u,j]:=\Sigma_{k=1}^{j}Pi[u,k]$, $1\leq j\leq n$ (Lemma 3.3(ii));

The pendants we select from each tree are those whose PRS values (ranks) equal to 1 or 2. We proceed to label the isolated vertices and the selected pendants as follows.

For pendants, we compute:

$label[u,v]:=$ if $((PRS[u,v]\leq 2)\wedge Pi[u,v])$

then $PRS[u,v]$

else 0;

The above statement should be interpreted as: if v is a pendant of the tree represented by u and its rank in that tree ($PRS[u,v]$) is 1 or 2, then label v with its rank else label v with 0.

For isolated vertices, we compute:

$label[u,u]$:= **if** $(PRS[u,u]=1) \wedge (degree[u,u]=0)$

then -1;

This completes Step 1.

In Step 2, we rank the trees in G_0 and pass the rank of each non-trivial tree to its two selected pendants. Recall that $Rep[k,v]=Rep[k,k]=1\ \forall k,v \epsilon V$ in Step 1.

$rank[i,i] := \Sigma_{k=1}^{i} Rep[k,i]$;

Clearly, $rank[i,i]$ is the rank of the tree represented by i in G_0.

We then pass the rank of each non-trivial tree to its two pendant vertices labelled with 1 or 2.

(Broadcast rowwise:) $rank[k,u]=rank[k,k]\ \forall k,v \epsilon V$;

$rank[v,v] := \Sigma_{k=1}^{n}$ **if** $Pi[k,v]$ **then** $rank[k,v]$ **else** 0;

At this stage, $rank[u,u]=rank[v,v]$ iff u,v are the two pendants in the same tree in G_0.

In Step 3, we construct the matrix A_1 such that $A_1[u,v]=1$ iff $(u,v) \epsilon A_1$:

Broadcast all the $label[u,v]$'s with value -1, 1 or 2 columnwise. Since there is at most one nonzero $label$ in

each column, this broadcasting can be realized as follows:

```
$label'[u,v] := \Sigma_{k=1}^{n} label[k,v]$;
(Broadcast rowwise:) $label[u,w] := label[u,u]$;
(Broadcast columnwise:) $rank'[w,u] := rank[u,u]$;
(Broadcast rowwise:) $rank[u,w] := rank[u,u]$;
```

Finally, execute the following statement locally at every processor:

```
if {($PRS'[u,v]=-1$) (i.e. $v$ is an isolated vertex) and
        the ranks of $u$ and $v$ differ in only 1 and $u$ is
        labelled with 1 or 2} or {$u$ and $v$ are both
        selected pendants but the rank of $u$ is 1 greater
        than that of $v$ while its label is 1 less than that
        of $v$ or the reverse}
then $A_1[u,v] := 1$
else $A_1[u,v] := 0$;
```

The above conditions can be easily tested by retrieving the contents of the *label*, *label'*, *rank* and *rank'* registers in each processor.

From Lemmas 3.1, 3.2, 3.3, 3.5 and Theorem 3.6, we have the indicated time and hardware resource complexities. ■

Lemma 3.12: Given an adjacency matrix T of a directed tree whose diameter is d, labelling the vertices of the tree with preorder numbers can be done in $O(t(n)*(\lg d+1))$ time with $H(n)$ hardware resources on the *MMM*.

Proof: We proceed in 3 steps.

In step 1, we compute $nd(v)$, the number of descendants of v, for every v. This is easily accomplished by first computing T^* and then adding all the 1's in each row of T^*. Recall that $T^*[u,k]=1$ iff k is a descendant of u.

$\therefore nd[u,v]:=\Sigma_{k=1}^{n} T[u,k]^*$.

In step 2, we compute $nds(v)$, the sum of the descendants of all elder brothers of v. Note that the sons of every vertex are ranked by their vertex numbers:

(Broadcast columnwise:) $nd'[u,j]:=nd[j,j]$;

Compute the sum of all the descendants of the sons of u whose vertex number are less than j. Note that j may not be a son of u here:

```
$nds[u,j]:= \Sigma_{k=1}^{j-1}$(if $T[u,k]$
                    then $nd'[u,k]$
                    else 0), (Lemma 3.3(ii));
```

Now, set $nds[u,j]$ to zero if j is not a son of u.

$nds[u,j]:=$ **if** $T[u,j]$ **then** $nds[u,j]$ **else** 0;

Finally, in step 3, we compute, $pre(v)$, the preorder number using the formula given in Lemma 2.15.

(Broadcast nds columnwise:) $nds'[w,j]:=nds[u,j]$, where $F(j)=u$; since every j has a single father, there is exactly one non-zero nds in each column, $\therefore$ Lemma 3.3 can be applied here.

(Recall again that $T^*[k,v]=1$ iff k is an ancestor of v.)

$pre[v,v]:=\Sigma_k(nds'[v,k]*T^*[k,v])$;

then compute

$pre[v,v] := pre[v,v] + level[v,v]$ locally at every processor.

Clearly, $pre[v,v] = pre(v)$.

From Lemmas 3.1, 3.2, 3.3 and 3.5, we have the indicated time and hardware resource complexities. ■

Lemma 3.13: Given an adjacency matrix M of an undirected tree G whose diameter is d'', constructing the edge set A_2 to bridge-connect G can be done in $O(t(n)\cdot(\lg d''+1))$ time with $H(n)$ hardware resources.

Proof: Construct an adjacency matrix T for a (directed) BFS spanning tree of G and choose a vertex with degree greater than 1 as the root. Note that when $n=2$, there is no way to bridge-connect G without introducing parallel edges. It is therefore resonable to assume $n \geq 3$ and this implies that a vertex of degree greater than 1 must exist. This step effectively converts G into a directed tree whose root has at least two sons.

Next, compute the preorder numbers $pre(v)$, $\forall v \in V$ (Lemma 3.12). Note that $pre[v,v] = pre(v)$. Then find the pendants and sort them by preorder number:

$notleaf[u,u] := \bigvee_{k=1}^{n} T[u,k]$, (Lemma 3.3(i));

Recall that T is a directed tree, $\therefore$ $notleaf[u,u] = 0$ iff u is a pendant.

Erase the preorder number of all non-pendants:

$pre[u,u] :=$ **if** $\neg notleaf[u,u]$ **then** $pre[u,u]$ **else** 0;

Order the pendants by preorder numbers:

```
(Broadcast rowwise:) $pre[u,w]:=pre[u,u]$;
(Broadcast columnwise:) $pre'[w,u]:=pre[u,u]$;
$Rank\text{-}leaf[u,u]:=\Sigma_{k=1}^{n}(Pre[u,k]\leq pre'[u,k])$,
                                    (Lemma 3.3(i));
```

Since the root is a non-pendant, $Rank\text{-}leaf[u,u]<n$ if u is a pendant, and $Rank\text{-}leaf[u,u]=n$ if u is a non-pendant. As a result, the non-pendants can be eliminated easily:

```
$Rank\text{-}leaf[u,u]:=$ if ($Rank\text{-}leaf[u,u]<n$)
                          then $Rank\text{-}leaf[u,u]$
                          else 0;
```

Thus, the pendants are sorted by preorder number and $Rank\text{-}leaf[u,u]$ indicates the position of u in the sorted sequence.

Now, determine the total number of pendants:

```
$T\text{-}leaf[u,u]:=\Sigma_{k=1}^{n}(pre'[u,k]>0)$, (Lemma3.3(i));
```

∴ $T\text{-}leaf[u,u]$, $\forall u\in V$, contains the total number of perdants in the directed tree.

Finally, we construct A_2 as follows.

For all u : $Rank\text{-}leaf[u,u]>0$, compute the rank of v such that (u,v) is to be inserted into A_2:

```
$Partner\text{-}rank[u,u]:=$if ($Rank\text{-}leaf[u,u]\leq\lceil T\text{-}leaf[u,u]/2\rceil$)
                    then $Rank\text{-}leaf[u,u]+\lfloor T\text{-}leaf[u,u]/2\rfloor$;
```

Note that the division can be realized by 'left shifting one bit'. Finally,

```
(Broadcast rowwise:)
$Partner\text{-}rank[u,w]:=Partner\text{-}rank[u,u]$;
```

(Broadcast columnwise:)

$Rank\text{-}leaf'[w,u]:=Rank\text{-}leaf[u,u]$;

$A_2[u,v]:= (Rank\text{-}leaf'[u,v]=Partner\text{-}rank[u,v])$.

Up to this point, we have indeed constructed a 'directed' edge set A_2 rather than the desired 'undirected' edge set. In order to complete the construction of A_2, we may construct the transpose of the (directed) A_2 just constructed. This process is exactly the same as that described in Lemma 3.7, the discussion is thus omitted.

From Lemmas 3.1, 3.2, 3.3, 3.12 and Theorem 3.6, we have the indicated time and hardware resource complexities. ■

Theorem 3.14: The bridge-connectivity augmentation problem can be solved in $O(t(n)\cdot(\max(\lg d,\lg d'')+1))$ time with $H(n)$ hardware resources on the *MMM*, where d'' is the diameter of $G_0(V_0,E_0 \cup A_0)$.

Proof: Immediate from Lemmas 3.10, 3.11, 3.12, 3.13 and the observation that d'' can be greater than d. ■

3.9 The Biconnectivity Problem

In this section, we shall implement Algorithm Biconnect of Chapter 2 on the *MMM*. We shall determine the set of all separation vertices at the same time.

Theorem 3.15: Given an adjacency matrix M of an undirected graph G, the set of all separation vertices and biconnected

components in G can be determined in $O(t(n)\cdot(\max(\lg d,\lg d'')+1))$ time with $H(n)$ hardware resources on the *MMM*, where d'' is the diameter of G'' defined in Section 2.10.2.

Proof: We proceed in 5 steps.

Steps 1-3 are the same as Steps 1-3 of the algorithm for bridge-connectivity; their discussions are thus omitted. Recall that after step 3, $level[v,k] = level(v)$, $\ell\mathrm{HLCA}[v,v]=\ell\mathrm{HLCA}(v)$, $\forall v,k\in V$ and the adjacency matrix M' of $G-T$ is available.

In Step 4, we shall construct an adjacency matrix M'' for $G''(E',E'')$ and determine the connected components of G'':

```
(Broadcast rowwise:) $\ell\mathrm{HLCA}[v,k] := \ell\mathrm{HLCA}[v,v]$, $\forall v,k\in V$;
(Broadcast columnwise:) $level'[k,v]:=level[v,v]$, $\forall v,k\in V$;
(Broadcast columnwise:) $\ell\mathrm{HLCA}'[k,v]:=\ell\mathrm{HLCA}[v,v]$, $\forall v,k\in V$;
(Consider if $\mathrm{HLCA}(u)\lessdot v\underset{\cdot}{\leq}u$):
  $M''[u,v]:=(T')^*[u,v] \wedge (level'[u,v]>\ell\mathrm{HLCA}[u,v])$;
(Consider if $\mathrm{HLCA}(v)\lessdot u\underset{\cdot}{\leq}v$):
  $M''[u,v]:=M''[u,v]$
               $\vee(T^*[u,v]\wedge(level[u,v]>l\mathrm{HLCA}'[u,v]))$;
(Consider the non-tree edges):
  $M''[u,v]:=M''[u,v] \vee M'[u,v]$;
```

(* Note: Since each v uniquely determines $F(v)$, we conveniently use v to represent $(F(v),v)$ in the vertex set of G'' here *)

Now compute $(M'')^*$, this determines the connected components of G''.

In Step 5, we shall determine the set of all separation vertices of G. Recall that $F[v,v]=F(v)$, $\forall v \epsilon V$ after Step 1.

(Broadcast columnwise:) $F'[k,v] := F[v,v]$, $\forall v,k \epsilon V$. Compute the matrix *subroot* such that *subroot*$[v,v]$ contains the root of the subtree (of the BFS spanning tree) containing v:

$\forall\ v \neq r$: *subroot*$[v,v]$:=$(\min\leq)_k$\{if $(M")^*[v,k]$ **then** $F'[v,k]$\},

(Lemma 3.3(i));

Once *subroot* is computed, constructing the matrix *Spt* such that *Spt*$[v,v]=1$ iff v is a separation vertex of G and adding these separation vertices to the connected components of $G"$ to form the biconnected components of G should be straightforward. We omitt the details here.

By Theorem 3.6 and Lemmas 3.1, 3.2, 3.3, 3.5, and note that $d"$ can be of $O(n)$ even if $d<<n$, we have: the biconnectivity problem can be sovled in $O(t(n)*(\max(\lg d,\lg d")+1))$ time with $H(n)$ hardware resources on the *MMM*.■

3.10 Performance on Existing Models

The aim of this section is to enhance the results we have achieved in the previous sections by showing that the *MMM* includes many of the well-known existing computer models as its special cases and that the performance of our algorithms on all these models are very efficient.

Lemma 3.16: The following computer models are instances of

the *MMM*:

MCN(VLSI) : Mesh connected Networks[CANN69,DEKE81,ATAL82];
PSN(VLSI) : Perfect Shuffle Networks[STON71,DEKE81];
CCC(VLSI) : Cube-connected Cycles[PREP81];
OTN(VLSI) : Orthogonal Tree Networks[NATH82];
OTC(VLSI) : Orthogonal Tree Cycles[NATH82];
SIMD-CCC : SIMD Cube-connected Computers[DEKE81];
PRAM : SIMD Shared Memory model with read conflicts permitted[WYLL79];
WRAM : SIMD Shared Memory Model with read and write conflicts permitted[SHIL82,KUCE82].

Proof: We show in the following table that each of these models has an ordinary matrix multiplication algorithm. For other features of the *MMM* that they possess, we refer the reader to the references cited.

The time and hardware resource complexities of the ordinary matrix multiplication algorithms on the above-listed models.

model	time	chip area	AT^2	# of processors	
MCN(VLSI)	$O(n)$	n^2	$O(n^4)$	---	[DEKE81]
PSN(VLSI)	$O(\lg^2 n)$	$n^6/\lg^3 n$	$O(n^6 \cdot \lg n)$	---	[DEKE81]
CCC(VLSI)	$O(\lg^2 n)$	$n^6/\lg^2 n$	$O(n^6 \cdot \lg^2 n)$	---	[PREP81]
OTN(VLSI)	$O(\lg^2 n)$	$n^4 \lg^2 n$	$O(n^4 \cdot \lg^6 n)$	---	[NATH81]
OTC(VLSI)	$O(\lg^2 n)$	n^4	$O(n^4 \cdot \lg^4 n)$	---	[NATH81]
SIMD-CCC	$O(\lg^2 n)$	---	---	$\lceil n^3/\lg n \rceil$	[DEKE81]
PRAM	$O(\lg^2 n)$	---	---	$\lceil n^3/\lg n \rceil$	[SAVA77]
WRAM	use the algorithm for the PRAM.				

The theorem thus follows. ■

Before combining Lemma 3.16 with the results obtained in the previous sections to produce the desired results, we would like to point out that the time complexity of our algorithms are dominated by the all-pair shortest path algorithm which is used to generate the BFS spanning forest.

If for a particular *MMM*, there exists an all-pair shortest path algorithm which runs faster than our algorithm described in Lemma 3.4, then that all-pair shortest path algorithm could be used in place of ours and the time complexity of the resulting algorithms is improved. This is the case for the MCN and WRAM as is shown below.

The time and hardware resource complexities of our algorithms on various existing models.

model	time	chip area	AT^2	# of processors	
MCN(VLSI)	$O(n)$†	n^2	$O(n^4)$	---	[VANS80]
PSN(VLSI)	$O(\lg n*L)$	$n^6/\lg^3 n$	$O(L^2*n^6/\lg n)$	---	
CCC(VLSI)	$O(\lg n*L)$	$n^6/\lg^2 n$	$O(n^6*L^2)$	---	
OTN(VLSI)	$O(\lg n*L)$	$n^4\lg^2 n$	$O(n^4\lg^4 n*L^2)$	---	
OTC(VLSI)	$O(\lg n*L)$	n^4	$O(n^4\lg^2 n*L^2)$	---	
SIMD-CCC	$O(\lg n*L)$	---	---	$\lceil n^3/\lg n\rceil$	
PRAM	$O(\lg n*L)$	---	---	$\lceil n^3/\lg n\rceil$	
WRAM	$O(L)$†	---	---	n^4	[KUCE82]

note: L = **max**$(\lg d,\lg d")+1$ $,1\leq d,d"\leq n$ for bridge-connectivity augmentation and biconnectivity;
= $\lg d+1$, otherwise.
† indicates the all-pair shortest path algorithm in the cited reference is used instead of Lemma 3.4.

Of all the above-mentioned models, no algorithms for the bridge-connectivity augmentation problem were known previously. Furthermore, with the exception of the MCN and PRAM, no algorithms for the bridge-connectivity and biconnectivity problems were reported. For the sake of comparison, we list all the previously known results below:

The time and hardware resource complexities of the previously known algorithms on various existing models.

model	time	chip area	AT^2	# of processors	
(i) The BFS spanning forest					
MCN(VLSI)	$O(n)$	n^2	$O(n^4)$	---	[ATAL82]
PSN(VLSI)	$O(\lg^2 n)$	$n^6/\lg n$	$O(n^6\lg^3 n)$	---	[DEKE81]
SIMD-CCC	$O(\lg^2 n)$	---	---	$\lceil n^3/\lg n\rceil$	[DEKE81]

(ii) The lowest common ancestors					
PRAM	$O(\lg^2 n)$	---	---	n^3	[SAVA81]
(iii) The fundamental cycles					
PRAM	$O(\lg^2 n)$	---	---	n^3	[SAVA81]
(iv) The 2-colorability (Bipartite)					
MCN	$O(n)$	n^2	$O(n^4)$	---	[ATAL82]
(v) Bridge-connectivity and Biconnectivity					
MCN(VLSI)	$O(n)$	n^2	$O(n^4)$	---	[ATAL82]
PRAM	$O(\lg^2 n)$	---	---	$\lceil n^3/\lg n \rceil$	[SAVA81]
or	$O(\lg^2 n \lg K)$‡	---	---	$\|E\|n+n^2\lg n$	[SAVA81]
or	$O(\lg^2 n)$	---	---	$n^2 \cdot \lg n$	[SAVA81]†

Note:† for bridge-connectivity only;
‡ K is the number of biconnected components in the graph.

The efficiency of our algorithms should be evident from the tables.

Finally, we shall prove a lemma which would be useful in employing existing results to improve the performance of our algorithms on the PRAM and the WRAM.

Lemma 3.17: Converting an undirected forest into an inverted (or directed) forest takes $O(\lg n)$ time with n^3 processors on the PRAM and the WRAM.

Proof: We shall find a directed spanning forest for the undirected forest using the all-pair shortest path method described in Lemma 3.6. Since there is a unique path between every pair of vertices in the undirected forest, only $O(1)$ time is required in each of the $O(\log n)$ iterations if n^3 processors are used. Specifically, this is accomplished as follows: Assign n processors to each pair of vertices u and v such that each of these n processors is attached to a

distinct vertex. During the ith iteration of executing the all-pair shortest path algorithm, the processor attached to vertex, say k, will examine the entries $D^{2**(i-1)}[u,k]$ and $D^{2**(i-1)}[k,v]$. If both of their values are finite and $D^{2**(i-1)}[u,k]=2^{i-1}$, then that processor will add their values and store the sum into $D^{2**i}[u,v]$. It is easily verified that there is exactly one such processor finding the above condition satisfied, hence no write conflicts would occur on the PRAM. ∎

As the first application of Lemma 3.17, we shall show that the processor bound of our algorithms on the WRAM can be improved to $O(n^3)$.

Corollary 3.18: All of our algorithms described in this Chapter run in $O(\lg n)$ time using n^3 processors on the WRAM.
Proof: Construct a minimum spanning forest for the given graph in $O(\lg n)$ time with $n+2|E|$ processors[AWER83]. Convert the minimum spanning forest into a directed forest using Lemma 3.17. It is easily verified that the remaining steps all take no more than $O(\lg n)$ time and n^3 processors. ∎

3.11 Conclusions

In contrast to sequential computation where the sequential RAM is chosen as an universally accepted model, there is no universally accepted model in parallel computation. Up to the present, the parallel computer model which has had the greatest degree of popularity is the PRAM.

This is due to its powerful fan-out capability which provides a means by which all the physical constraints inherent in the interconnection network are bypassed. The designer can thus concentrate on uncovering the inherent data-dependency of the given problem. This makes the task of algorithm design much easier. As a consequence, parallel algorithms published in the literature are mostly designed for the PRAM or its stronger version, the WRAM. Unfortunately, this fan-out capability is unrealistically powerful in the sense that it cannot be realized with current technology. Its acceptability as a universal model is questionable.

The restricted models which take the technological constraints under consideration are preferable from the practical point of view since they are well-suited for current VLSI technology. However, the constraints imposed by their limited fan in/out capability tend to obscure the designer's insight and make the design of efficient algorithms more difficult. Furthermore, portability between these models is weaker due to the vast variety of ways of constructing the interconnection network. To remedy the first drawback, one may first design an algorithm for the problem on the PRAM and then map the algorithm onto the restricted model at hand. In fact, some work has been done using this approach[SCHW80,VISH81b]. However, a degradation in time complexity (at least a factor of lg$n$ in the existing works) has always been induced. To remedy the second

problem, we may simulate one model on the other. So far, these simulations are done at the abstract level. [SIEG77] and [SIEG79] are exceptions.

In our opinion, the *MMM* model proposed in this chapter provides a better solution to the above problems. By reducing many of the basic operations we use into operations of the form defined in Lemma 3.1, we have managed to demonstrate that the algorithms presented in Chapter 2 for the PRAM can be implemented on many of the existing restricted models with no degradation in time. Moreover, the portability of these algorithms on various models is immediate — no tedious simulation is necessary. Thus, the *MMM* model seems to be a promising tool for designing portable algorithms.

Chapter 4

IMPLEMENTATION ON THE SEQUENTIAL RAM

4.1 Introduction

In Chapter 1, it was mentioned that given a parallel algorithm for the PRAM, if the algorithm runs in $T(n)$ time using $P(n)$ processors, then the same algorithm can run on the sequential RAM in $T(n)\cdot P(n)$ time. An implication of this observation is that each of the algorithms presented in Chapter 2 immediately induces an $O(n^2)$ or $O(n^2 \lg n)$ time algorithm for the sequential RAM. Although this result is optimal for dense graphs, we shall show that, we can do better for sparse graphs for some of the problems. In this chapter, we present a sequential version of Algorithm Biconnect which finds all the biconnected components as well as all the separation vertices of an undirected graph. This algorithm requires $O(n+|E|)$ time and space which is optimal for all graphs. Moreover, it does not rely on the well-known depth-first search spanning tree but uses *any* spanning tree of the graph. Thus, this is another example to show that depth-first search is not always necessary for dealing with connectivity properties of graphs 'efficiently' (the first example was given by Tarjan in [TARJ74] concerning finding all bridges). It is also shown that this algorithm is a generalization of Tarjan's depth-first search algorithm presented in [TARJ72]. The algorithm also detects all bridges and hence the bridge-connected components of the

graph within the same time and space bounds.

We also present a general program scheme for the bridge-connectivity problem. This general program scheme runs on the sequential RAM in $\max(O(n+|E|),T(g,\emptyset_1,\emptyset_2))$ time and $\max(O(n+|E|),S(g,\emptyset_1,\emptyset_2))$ space, and on the PRAM in $\max(O(n/K+\lg^2 n),T(g,\emptyset_1,\emptyset_2))$ time with $nK(K\geq 1)$ processors, where g, $\emptyset_1$, $\emptyset_2$ are parameters of the general program scheme. Clearly, the optimality of the program scheme depends on the complexities of $T(g,\emptyset_1,\emptyset_2)$ and $S(g,\emptyset_1,\emptyset_2)$. We shall show that by substituting several appropriate functions for the parameters g, $\emptyset_1$ and $\emptyset_2$, we can derive most of the existing optimal sequential algorithms as well as new optimal parallel algorithms including Algorithm Bridges presented in Chapter 2 for finding the bridges.

4.2 The Sequential Algorithm for Biconnectivity

In this section, we present a sequential algorithm for finding all biconnected components and all separation vertices of an undirected graph. As with Chapter 2, since each biconnected component is completely determined by its vertex set, it suffices to find the vertex sets of all the biconnected components.

Let $G(V,E)$ be an undirected graph. Without loss of generality, we again assume that G is connected and $V=\{1,2,...,n\}$. We also use the function HLCA(u) defined in Chapter 2. However, we redefine it here because there is a slight modification involved.

Definition: Let $T(V,E')$ be a directed spanning tree of G and $u \in V$.

$$\text{HLCA}(u)=\text{LCA}(u,v) \text{ in } T, \text{ where } (u,v) \in E \text{ and}$$
$$\text{depth}(\text{LCA}(u,v)) \le \text{depth}(\text{LCA}(u,v')),\ \forall (u,v') \in E.$$

4.2.1 An Outline of the Algorithm

We give an outline of the algorithm below:

Algorithm Seq-biconnect:

1. Create a spanning tree T' of G;
2. Convert T' to a directed tree $T(V,E')$; again let the functions F and depth be such that F(v), depth(v) are the father and depth of v in T respectively, $\forall v \in V$;
3. Partition T into connected subgraphs, called **trimmed-subtrees** $\{T_i\}$ such that each of them has the following properties:

(i) Each T_i is a directed tree whose root has exactly one son;

(ii)a. let r_i be the root of a T_i, for any vertex $v \ne r_i$ in T_i, HLCA(v) is a descendant of r_i;

b. for every internal vertex $v \ne r_i$ in a T_i, there exists a proper descendant d of v for which HLCA(d) is a proper ancestor of v;

c. let l be a leaf-node of a T_i, then for every proper descendant d of l in T, HLCA(d) is a descendant of l;

4. Construct a graph $G''(V'',E'')$, such that $V''=\{T_i\}$ and $(T_k,T_m) \in E''$ iff there exists an edge e in E connecting T_k

and T_m and the end-vertices of e is neither the roots of the two T_i's. Find all the connected components $\{\{T_i\}_j\}$ in G'', then each $U_j\{T_i(V)\}_j$ is the vertex set of a biconnected component in G, and vice versa, where $T_i(V)$ is the vertex set of T_i. ∎

4.2.2 Partitioning the Directed Tree

The input to the algorithm is an adjacency list of G. Steps 1 and 2 are trivial and can clearly be done in $O(n+|E|)$ time and space. The resulting directed spanning tree T is represented by an adjacency list which takes $O(n)$ space.

To realize the partition $\{T_i\}$ of T in step 3, we will traverse the directed spanning tree T in preorder and label every vertex with its preorder number. Henceforth, we will name each vertex by its preorder number, i.e. $v=pre(v)$.

Definition: For $v \in V$,

$low(v)=\mathbf{min}\{w|w=\mathrm{HLCA}(x)\ x \text{ is a descendant of } v \text{ in } T\}$.

For example, in Figure 4.1(i), $low(3)=1$ and $low(15)=9$. Due to the associativity of **min**, the above equation can be rewritten as:

$low(v)=\mathbf{min}(\{\mathrm{HLCA}(v)\}U\{low(s)|s \text{ is a son of } v \text{ in } T\})$

The complete description of step 3 is as follows.

1. *precount*:=1; compute F(v), depth(v) $\forall v \in V$; compute

HLCA(v),$\forall v \in V$, using the off-line lowest common ancestors algorithm presented in [HARE80].

2. CreateTi(r), where r is the root of T.

```
procedure CreateTi(v);
begin
pre(v):=precount;
precount:=precount+1;
Push v on stack stackT;
low(v):=HLCA(v);
for every son s of v do
  begin
  CreateTi(s);
  if low(s)=pre(v)
    then pop stackT until s is popped and then output v
    else low(v):=min(low(v),low(s))
  end
end{of CreateTi};
```

An example of the result of executing step 3 on the graph in Figure 4.1(i) is given in Figure 4.1(ii).

Theorem 4.1: Step 3 correctly generates the set of all trimmed-subtrees $\{T_i\}$.

Proof: We want to prove that whenever $low(s)=v$, the vertices on *stackT* from s right up to the top plus vertex v constitute the vertex set of a T_i. This is done by induction on the number of T_i's in T.

If T has only one T_i, then the proof is trivial.

Assume that the induction hypothesis holds for all T having m T_i's. Consider a T having $m+1$ T_i's. Let CreateTi(s) be the first call of CreateTi ending with $low(s)=v$. This means no vertices have been popped from *stackT*. Therefore, the vertices on *stackT* from s to the top and vertex v form the vertex set of a subtree Tv of T rooted at v. Tv clearly

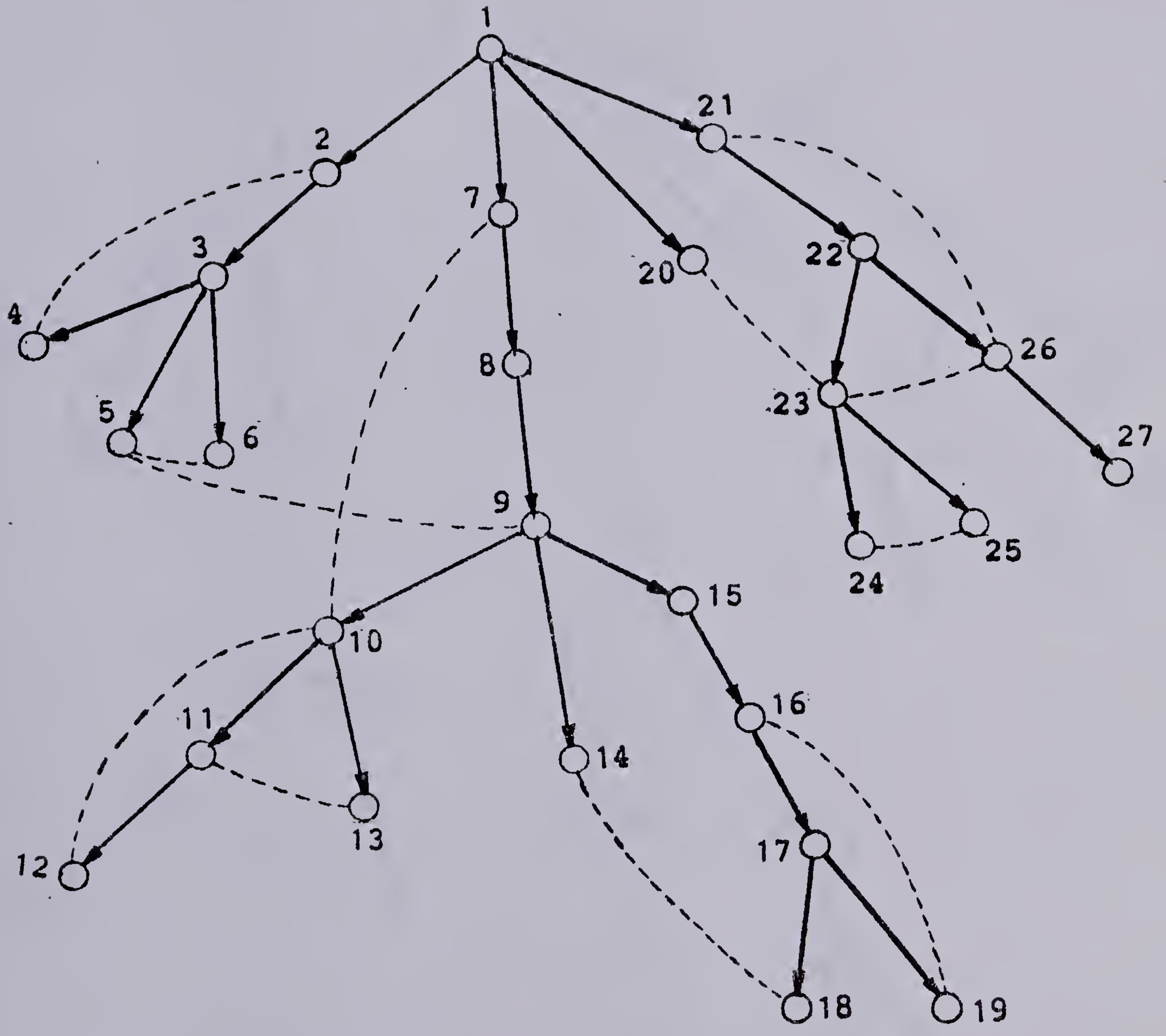

1
2
21
7
3
20
22
4
8
26
5
23
6
27
9
25
24
15
10
11
16
14
13
17
12
18
19

Figure 4.1(i)

A directed spanning tree *T*(*V*,*E*').

The solid lines are the tree edges.

The dotted lines are the edges in *G*-*T*.

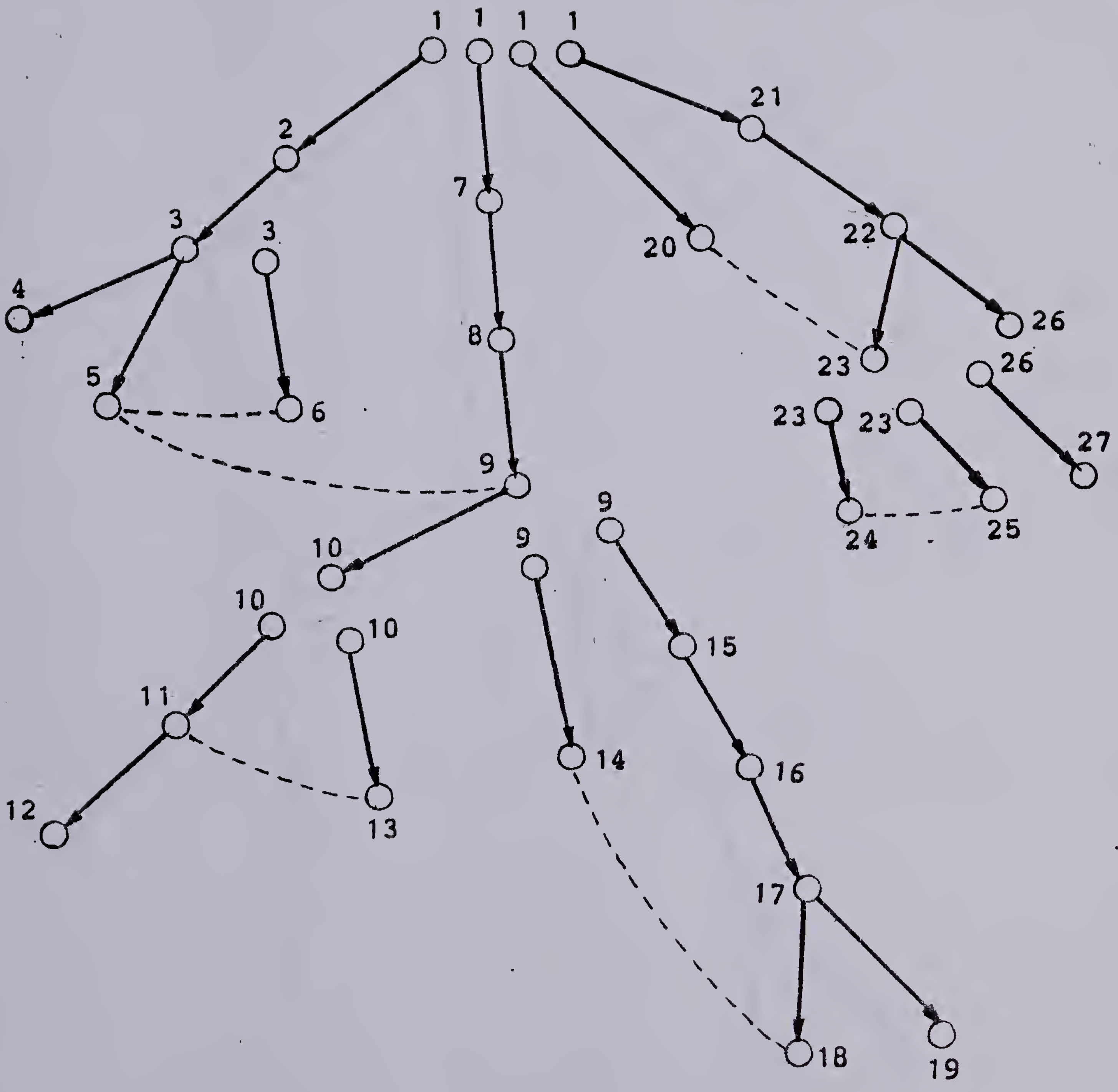

1
1
1
1
2
21
7
3
3
20
22
4
26
8
23
26
5
6
23
23
27
9
24
25
10
9
9
10
10
15
11
14
16
12
13
17
18
19

Figure 4.1(ii)

The partition $\{T_i\}$ of T.

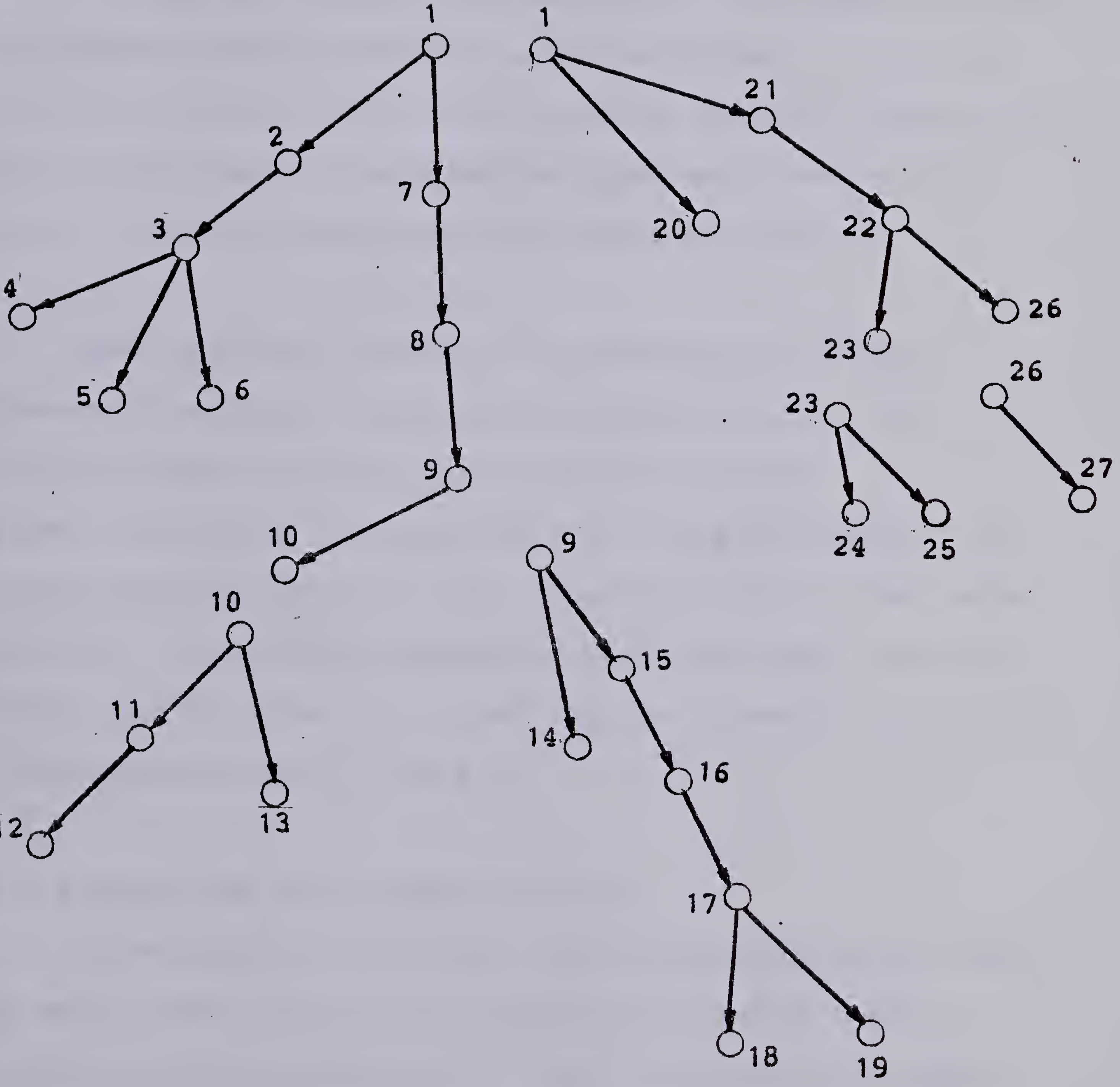

1
1
2
21
7
3
20
22
4
26
8
23
5
6
26
23
9
27
10
9
24
25
10
15
11
14
16
12
13
17
18
19

Figure 4.1(iii)

The partition $\{BC_j\}$ of T.

possesses properties (i) and (ii)c. $low(s)=v$ implies that Tv possesses property (ii)a. Finally, Tv must possess property (ii)b for otherwise there is a proper descendant w of s for which $low(w_s)=w$ where w_s is a son of w. This contradicts the assumption that CreateTi(s) is the first call ending with $low(s)=v$. Thus, Tv is a T_i of T. After removing the vertices in Tv from *stackT*, the induction hypothesis ensures that step 3 correctly generates the remaining m T_i's.■

The complexity of Step 3 is analyzed as follows.

Theorem 4.2: Step 3 of Algorithm Seq-biconnect takes $O(n+|E|)$ time and space on the sequential RAM.

proof: Traversing the spanning tree T and maintaining the stack *stackT* takes $O(n)$ time. Computing HLCA(v),$\forall v \in V$ takes $O(n+|E|)$ time and space[HARE80]. Moreover, both the stack *stackT* and the stack for governing the traversal of T do not grow beyond n unit of space. ■

4.2.3 Combining the Trimmed-subtrees

After step 3 is finished, the directed spanning tree T is partitioned into trimmed-subtrees T_i's. From Lemma 2.25(i),(ii) and property (ii) of T_i, it is easily shown that each T_i is contained within a unique biconnected component in G. It is also easily shown that every two adjacent T_i's intersect at no more than one vertex. If, however, two adjacent T_i's are connected by an edge in $G-T$, which is not incident with either of the roots of them, then

they should be combined together as they are contained within the same biconnected component(Lemma 2.25(iii)). In the following, we will show that when no such combination can be carried out any further, the result is the vertex sets of all the biconnected components in G.

As an example, consider Figure 4.1(ii) again. For clarity, we denote each trimmed-subtree in the figure by $T_{s(i)}$ where $s(i)$ is the preorder number of the *unique* son of the root of the trimmed-subtree. For instance, the trimmed-subtree containing vertices 9, 15, 16, 17, 18, and 19 is denoted by T_{15}. Hence the directed tree in Figure 4.1(i) is divided into trimmed-subtrees T_2, T_7, T_{20}, T_{21}, T_6, T_{11}, T_{13}, T_{14}, T_{15}, T_{24}, T_{25} and T_{27} in Figure 4.1(ii). T_2 and T_7 are connected by an edge (5,9) in $G-T$ and neither 5 nor 9 is the root of T_2 or T_7. It can be easily seen that T_2 and T_7 are indeed contained within the same biconnected component. Similarly, the edge (5,6) joining T_2 and T_6 implies that T_2 and T_6 are contained within the same biconnected component. Consider again the edge (5,9); this edge also connects T_2 and T_{15}. However, 9 being the root of T_{15} does not imply that T_2 and T_{15} are contained within the same biconnected component (in fact, they are not). The same argument applies to the edge (23,26) which connects T_{24} and T_{27}, and the edge (12,10) which connects T_{11} and T_7.

Definition: Let $T_1, T_2 \epsilon \{T_i\}$ be two trimmed-subtrees. $\mathbf{T_1 — T_2}$ iff (i) $T_1 = T_2$;

or (ii) there exists an edge e in $G\text{-}T$ such that e connects T_1 and T_2, and e is not incident with r_1 or r_2, where r_1 and r_2 are the roots of T_1 and T_2 respectively,

or (iii) $T_1 — T_3$ for some $T_3 \epsilon \{T_i\}$ such that $T_3 — T_2$.

It can be easily shown that any edge in $G\text{-}T$ violating the criteria given in the above definition is of one of the types depicted in Figure 4.2.

The binary relation — is an equivalence relation on $\{T_i\}$ and thus partitions $\{T_i\}$ into equivalence classes $\{\{T_i\}_j\}$. Let $BC_j = U_j\{T_i\}_j$. The following theorem points out that the vertex sets of all the BC_j's is exactly the vertex sets of all the biconnected components in G.

Theorem 4.3: v,v' are in B, for some biconnected component B of G, iff $v,v' \epsilon BC_j(V)$, for some j, where $BC_j(V)$ stands for the set of all vertices in BC_j.

Proof: If part: From Lemma 2.25(iii), it is obvious that each BC_j is completely contained within a biconnected component of G.

Only if part: This is proven by contradiction. Without loss of generality, let us assume that BC_k and BC_m are distinct, and $BC_k(V) U BC_m(V) = B(V)$. It should be clear that BC_k and BC_m intersect at no more than one vertex which is either the root of BC_k or BC_m. Without loss of generality, we assume they intersect at r_k, the root of BC_k. Since r_k cannot be a separation vertex in B, there must be an edge in B joining

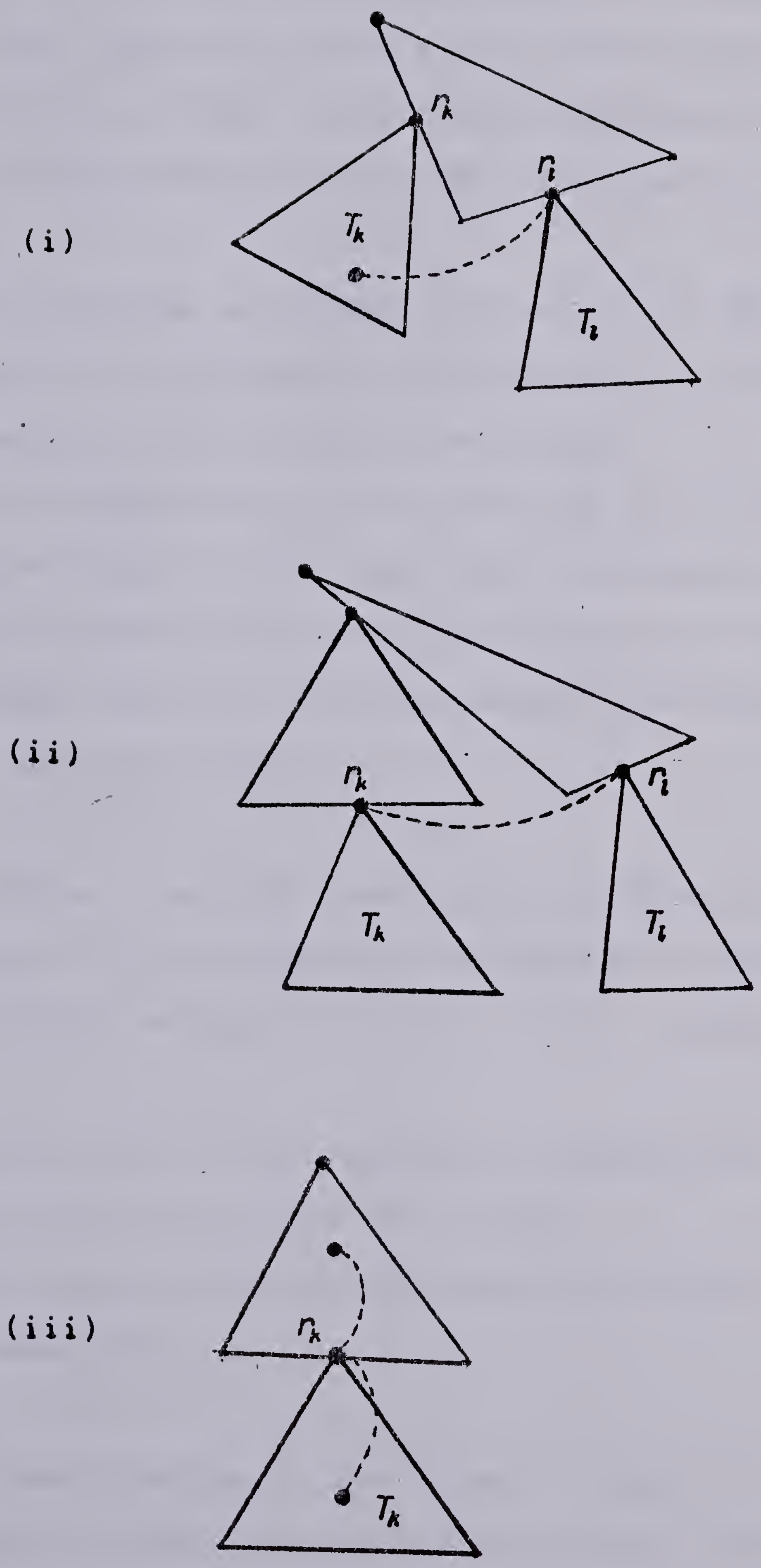

(i)
r_k
r_l
T_k
T_l
(ii)
r_k
r_l
T_k
T_l
(iii)
r_k
T_k

Figure 4.2 Non-tree edges violating the — definition

BC_k and BC_m not incident with r_k. Obviously, this edge is not incident with r_m either, for otherwise, r_m would be in T_k forcing $r_k = r_m$ which contradicts the fact that the edge is not incident with r_k. This implies that the two BC_j's would have the '—' relationship leading to a contradiction.■

Lemma 4.4: The problem of finding the set of all BC_j's in T can be reduced to the problem of finding the set of all connected components of an undirected graph.

Proof: Define a graph $G''(\{T_i\}, E'')$ such that $(T_k, T_m) \in E''$ iff there exists an edge e in $G-T$ such that e connects T_k and T_m and e is not incident with either of the roots of the two T_i's. It is clear that T_k, T_m belong to the same connected component of G'' iff T_k—T_m.■

It should be clear that every BC_j is a directed spanning tree of its corresponding biconnected component. In fact, $\{BC_j\} = \{B_j \wedge T\}$ defined in Section 2.10.3. Consequently, we have:

Theorem 4.5: Let $a \in V$. a is a separation vertex of G

iff a is the root of some BC_j if $a \neq r$;

or a is the root of more than one BC_j if $a = r$.

Proof: See Lemma 2.29.■

Each trimmed-subtree T_i determined in step 3 is represented by a linear list containing all the vertices of T_i except the root r_i. The reason for excluding r_i should be

obvious as it may belong to other T_i's at the same time. Nevertheless, r_i can be relocated easily as $r_i=F(s_i)$ where s_i is the 'only' son of r_i in T_i. s_i is also used as the representative of T_i in G''. The linear lists are created while the vertices are popped from *stackT* in procedure CreateTi. Note that s_i is the last vertex popped from the stack and is therefore easily identified. A vector *superV* is also created at the same time such that for each vertex v in G, $superV(v)=s_i$ iff v is in T_i and $v \neq r_i$. The purpose of *superV* is to tell to which T_i each vertex v belongs. The exclusion of r_i from the list ensures that situations depicted in Figure 4.2(i),(ii) are always handled correctly. In other words, the edges shown would never be mistaken as edges establishing the —— relationship between the T_k and T_m shown. As a consequence, the edges which must be taken care of are those edges in which one end-vertex is a r_k and the other end-vertex is in T_k (Figure 4.2(iii)).

To create the graph $G''(V'',E'')$ in step 4, an adjacency list of G'' must be created. We proceed as follows. The linear lists for the T_i's are scanned one at a time. Suppose the linear list being examined corresponds to T_k, then for each vertex v stored in the linear list, the adjacency list of v in G is scanned. For each node u encountered in the adjacency list, the following tests are performed:

(i) test if $superV(u) \neq s_k$;

(ii) test if $F(s_k) \neq u$;

(iii) test if $F(superV(u)) \neq v$.

Test (i) is to ensure u,v do not belong to the same trimmed-subtree T_k while tests (ii) and (iii) are done to ensure that the edge (v,u) is not an edge of the form shown in Figure 4.2(iii). Note that no tree edges pass the tests. If the edge passes the tests, then a new node containing the vertex $superV(u)$ is added to the adjacency list of s_k in G'', thereby establishing the '—' relationship between s_k and $superV(u)$. When all the linear lists are processed, the adjacency list of G'' is complete. Note that G'' may be a multigraph (i.e. there may be more than one edge joining two vertices). However, $|E''| \leq |E|$.

The connected components of G'' are then determined by traversing the graph G'', using any standard traversal technique. For each connected component of G'', all the linear lists corresponding to the T_i's in the component are merged together and the root r_j of smallest depth among all the roots of these T_i's is determined. This r_j and the vertices in the list resulting from the merge form the vertex set of a biconnected component in G. Moreover, from Theorem 4.5, the r_j is a separation vertex of G if $r_j \neq r$. To determine if the root r is a separation vertex of G, we proceed as follows. A Boolean variable called $once$ is initialized to **false** at the beginning. Whenever a component of G'' is completely traversed, the corresponding vertex r_j is examined. If it is r, the variable $once$ is examined. If $once$ has the value **false**, it will be set to **true** and the r is discarded. Otherwise, by Theorem 4.5, r must be a

separation vertex of G.

When finally all the components of G'' are determined, the vertex sets of all the biconnected components as well as the separation vertices of G are also determined. Figure 4.1(iii) illustrates how the trimmed-subtrees depicted in Figure 4.1(ii) are combined to form the BC_j's. The correctness of step 4 should be obvious from the above dicussion. The time complexity of step 4 is analyzed as follows.

Theorem 4.6: Step 4 of Algorithm Seq-biconnect takes $O(n+|E|)$ time and space.

Proof: The construction of the adjacency list of G'' takes $O(n+|E|)$ time. Traversing G'' so as to determine the vertex sets of all the biconnected components of G takes $O(n+|E|)$ time and the creation of the vector *superV* takes $O(n)$ time. As for the space complexity, *superV* takes $O(n)$ space and The adjacency list of G'' is cleary bounded by $O(n+|E|)$ even if G'' is a multigraph. Hence, step 4 can be done in $O(n+|E|)$ time and space. ■

In summary, the Algorithm Seq-biconnect takes $O(n+|E|)$ time and space to generate the vertex sets of all the biconnected components and the set of all separation vertices of G.

4.2.4 Discussion of Other Related Work

Consider what happens if the directed spanning tree T happens to be a depth-first search spanning tree of G. In this case, no cross edges[TARJ72] exist. This implies that the '—' relationship does not exist between any two T_i's. Thus step 4 will be omitted. As for step 3, since all the edges (v,u) in $E-E'$ are back edges[TARJ72], LCA$(v,u)=u$ or v depending on which is the ancestor of the other. As a consequence, the value $low(v)$ becomes:

$$low(v)=\min(\{F(v)\} \cup \{low(s) \mid s \text{ is a son of } v\} \cup \{w \mid (v,w) \text{ is a back edge in } G-T\}).$$

Comparing $low(v)$ with $lowpt(v)$ in [TARJ72,p.151] and procedure CreateTi with procedure BICONNECT in [TARJ72,p.153], it is obvious that they are basically equivalent. Hence, the depth-first search algorithm for determining the biconnected components[TARJ72] is a special case of our algorithm. Clearly, our sequential algorithm could also detect the bridges and hence the bridge-connected components of G within the same time and space bounds.

Remark:

Recently, Tarjan has independently achieved a similar result[TARJ82] by using another technique which does not involve computing the LCA values. His algorithm is not a generalization (in the sense described above) of the depth-first search[TARJ72] algorithm.

4.3 A General Program Scheme for Finding Bridges

4.3.1 The General Program Scheme

In this section, we present a general program scheme for finding the bridges of an undirected graph $G(V,E)$. We shall show that by substituting the parameters in the program scheme with various specific functions, a number of optimal algorithms for finding the bridges can be derived. Included in these are the known optimal sequential algorithms as well as new parallel algorithms for finding the bridges.

The general program scheme is based on the following lemma which was stated in a different way in Theorem 2.20.

Lemma 4.7: Let $T(V,E')$ be a directed spanning tree of a connected, undirected graph $G(V,E)$ and $e=<F(a),a>\epsilon E'$. e is a bridge in G iff for every descendant v of a, if $(v,w)\epsilon E-E'$, then w is a descendant of a.

The General Program Scheme.
Input: The adjacency matrix or list of $G(V,E)$;
Output: The set of all bridges of $G(V,E)$;

```
1. Find a directed spanning tree $T(V,E')$ of $G(V,E)$;

2. Define $g:(E-E')U E''\longrightarrow N$, $\emptyset_1:V\longrightarrow N$, $\emptyset_2:V\longrightarrow N$, where $E''$ is the
   set $\{(v,v)|v\epsilon V\}$, $N$ is the set of integers, such that the
   following condition is satisfied:
   for every $a\epsilon V$, let $v$ be any descendant of $a$, and
   $(v,w)\epsilon(E-E')U\{(v,v)\}$,
      then $\emptyset_1(a)\leq g(v,w)\leq\emptyset_2(a)$ iff $w$ is a descendant of $a$.
```

3. For every $v \in V$, find
$\mathrm{L}(v)=\mathbf{min}\{g(v,w) \mid (v,w) \in (E-E')U\{(v,v)\}\}$;
$\mathrm{H}(v)=\mathbf{max}\{g(v,w) \mid (v,w) \in (E-E')U\{(v,v)\}\}$.

4. For every $a \in V$, find
$min(a)=\mathbf{min}\{\mathrm{L}(v) \mid v \text{ is a descendant of } a \text{ in } T\}$;
$max(a)=\mathbf{max}\{\mathrm{H}(v) \mid v \text{ is a descendant of } a \text{ in } T\}$.

5. For every $a \in V$,
$(\mathrm{F}(a),a)$ is a bridge iff $\phi_1(a) \le min(a)$ and $max(a) \le \phi_2(a)$.

Theorem 4.8: The general program scheme correctly finds all the bridges of $G(V,E)$.

Proof: From the definitions of $min(a)$ and $max(a)$,

$\phi_1(a) \le min(a) \le max(a) \le \phi_2(a)$

iff $\phi_1(a) \le \mathrm{L}(v)$ and $\mathrm{H}(v) \le \phi_2(a)$ $\forall v \in V$, where v is a descendant of a

iff $\phi_1(a) \le g(v,w) \le \phi_2(a)$, $\forall (v,w) \in (E-E')U\{(v,v)\}$ and v is a descendant of a

iff for every descendant v of a, if $(v,w) \in (E-E')U\{(v,v)\}$, then w is a descendant of a (The condition given in Step 2)

iff $(\mathrm{F}(a),a)$ is a bridge in G (Lemma 4.7). ■

4.3.2 Implementation on the Sequential RAM

Theorem 4.9: The general program scheme takes $\mathbf{max}(O(n+|E|),\mathrm{T}(g,\phi_1,\phi_2))$ time and $\mathbf{max}(O(n+|E|),\mathrm{S}(g,\phi_1,\phi_2))$ space to find the set of bridges on the sequential RAM, where $\mathrm{T}(g,\phi_1,\phi_2)$ and $\mathrm{S}(g,\phi_1,\phi_2)$ are the time and space needed to compute the functions g, ϕ_1 and ϕ_2.

Proof: Using the adjacency list of G, Steps 1 and 3 can

clearly be done in $O(n+|E|)$ time and space. Due to the associativity of **min**, $min(a)=\mathbf{min}(\{min(a_s)|a_s$ is a son of $a\}U\{\mathrm{L}(a)\})$. The same argument applies to $max(a)$. Therefore by simply traversing the spanning tree T in preorder or postorder, Step 4 can be done in $O(n+|E|)$ time and space. Step 5 takes $O(n)$ time and space. Hence the general program scheme takes $\mathbf{max}(O(n+|E|),\mathrm{T}(g,\emptyset_1,\emptyset_2))$ time and $\mathbf{max}(O(n+|E|),\mathrm{S}(g,\emptyset_1,\emptyset_2))$ space. ■

Based on the above general program scheme, several optimal sequential algorithms for finding the bridges of G can be generated as follows.

Corollary 4.10: Let $g(v,w)=pre(w)\ \forall(v,w)\epsilon(E-E')U\{(v,v)\}$;

$\emptyset_1(a)=pre(a)$;

$\emptyset_2(a)=pre(a)+nd(a)-1$, $\forall a\epsilon V$, where $nd(a)$ is the number of descendants of a and $pre(a)$ is the preorder number of a. Then the general program scheme finds the bridges of $G(V,E)$ in $O(n+|E|)$ time and space.

Proof: It is easy to show that for every $a\epsilon V$, if v is a descendant of a and $(v,w)\epsilon(E-E')U\{(v,v)\}$ then w is a descendant of a iff $pre(a)\leq pre(w)\leq pre(a)+nd(a)-1$. Therefore the resulting program scheme correctly identifies all the bridges. Furthermore, $pre(v)\ \forall v\epsilon V$ can be computed in $O(n)$ time and space[HORO79]. $nd(v)\ \forall v\epsilon V$ can be computed in $O(n+|E|)$ time and space by using the fact that $nd(v)=\Sigma_i nd(i)+1\ \forall i\epsilon V_s$ where V_s is the set of all sons of v.

Hence $T(g,\emptyset_1,\emptyset_2)=S(g,\emptyset_1,\emptyset_2)=O(n+|E|)$. ∎

It is interesting to note that when T is a depth first search spanning tree, Corollary 4.10 is equivalent to the depth first search algorithm for finding the bridges[EVEN79,p.67, Ex.3.7].

Corollary 4.11: Let $g(v,w)=post(w)$, $\forall(v,w)\epsilon(E-E')U\{(v,v)\}$;

$\emptyset_1(a)=post(a)-nd(a)+1$;

$\emptyset_2(a)=post(a)$, $\forall a\epsilon V$, where $post(a)$ is the postorder number of a.

Then the general program scheme finds the bridges of $G(V,E)$ in $O(n+|E|)$ time and space.

Proof: It is easily proved that for every $a\epsilon V$, if v is a descendant of a and $(v,w)\epsilon(E-E')U\{(v,v)\}$ then w is a descendant of a iff $post(a)-nd(a)+1\leq post(w)\leq post(a)$. Moreover, since $post(v)$ $\forall v\epsilon V$ can be computed in $O(n)$ time and space[HORO79] and $nd(v)$ $\forall v\epsilon V$ can be computed in $O(n+|E|)$ time and space, $T(g,\emptyset_1,\emptyset_2)= S(g,\emptyset_1,\emptyset_2)=O(n+|E|)$. ∎

Note that this algorithm is equivalent to that of Tarjan[TARJ74].

Corollary 4.12: Let $g(v,w)=depth(\text{LCA}(v,w))$, $\forall(v,w)\epsilon(E-E')U\{(v,v)\}$, where $\text{LCA}(v,w)$ is the lowest common ancestor of v and w in T, $depth(a)$ is the depth of a in T;

$\emptyset_1(a)=depth(a)$;

$\emptyset_2(a)=n$ (note that $depth(v)\leq n$ $\forall v\epsilon V$), $\forall a\epsilon V$,

then the general program scheme finds all the bridges in $O(n+|E|)$ time and space.

Proof: It is easily proved that for every $a \in V$, if v is a descendant of a and $(v,w) \in (E-E')U\{(v,v)\}$ then w is a descendant of a iff $depth(a) \leq depth(\mathrm{LCA}(v,w)) \leq n$. Computing $\mathrm{LCA}(v,w)$ $\forall (v,w) \in (E-E')U\{(v,v)\}$ takes $O(n+|E|)$ time and space[HARE80] and computing $depth(v)$ $\forall v \in V$ takes $O(n)$ time and space. Hence $\mathrm{T}(g,\emptyset_1,\emptyset_2)=\mathrm{S}(g,\emptyset_1,\emptyset_2)=O(n+|E|)$. ■

4.3.3 Implementation on the PRAM

Theorem 4.13: The general program scheme takes **max**$(O(n/\mathrm{K}+\lg^2 n),\mathrm{T}(g,\emptyset_1,\emptyset_2))$ time with $n\mathrm{K}(\mathrm{K}\geq 1)$ processors to find the bridges of $G(V,E)$ on the PRAM, where $\mathrm{T}(g,\emptyset_1,\emptyset_2)$ is the time taken to compute (define) the functions g, $\emptyset_1$ and $\emptyset_2$ with $n\mathrm{K}(\mathrm{K}\geq 1)$ processors.

Proof: By Lemma 2.2, $L(v)$, $H(v)$, $min(a)$ and $max(a)$ $\forall v,a \in V$ can all be determined in $O(n/\mathrm{K}+\lg \mathrm{K})$ time with $n\mathrm{K}(\mathrm{K}\geq 1)$ processors. Step 5 clearly takes constant time and step 1 takes $O(n/\mathrm{K}+\lg^2 n)$ time with $n\mathrm{K}(\mathrm{K}\geq 1)$ processors(Theorem 2.5). Hence, the general program scheme takes $\max(O(n/\mathrm{K}+\lg^2 n),\mathrm{T}(g,\emptyset_1,\emptyset_2))$ time with $n\mathrm{K}(\mathrm{K}\geq 1)$ processors. ■

As with the sequential machines, optimal parallel algorithms can be derived from the general program scheme by using preorder, postorder and LCA on the PRAM.

Corollary 4.14: By defining g, $\emptyset_1$ and $\emptyset_2$ in one of the following ways, the general program scheme runs in

$O(n/K+\lg^2 n)$ time with $nK(K\geq 1)$ processors on the PRAM.

(i) Let $g(v,w)=pre(w)\ \forall(v,w)\epsilon(E-E')U\{(v,v)\}$;

$\emptyset_1(a)=pre(a)$;

$\emptyset_2(a)=pre(a)+nd(a)-1,\ \forall a\epsilon V$.

(ii) Let $g(v,w)=post(w)\ \forall(v,w)\epsilon(E-E')U\{(v,v)\}$;

$\emptyset_1(a)=post(a)-nd(a)+1$;

$\emptyset_2(a)=post(a)\ \forall a\epsilon V$.

(iii) Let $g(v,w)=depth(\mathrm{LCA}(v,w))\ \forall(v,w)\epsilon(E-E')U\{(v,v)\}$;

$\emptyset_1(a)=depth(a)$;

$\emptyset_2(a)=n$.

Proof: For (i) and (ii), $pre(v)$, $post(v)$ and $nd(v)\ \forall v\epsilon V$, can be computed in $O(n/K+\lg n)$ time with $nK(K\geq 1)$ processors(Lemma 2.15). For (iii), the resulting algorithm is Algorithm Bridges presented in Chapter 2. In any of these cases, we have $T(g,\emptyset_1,\emptyset_2)=O(n/K+\lg n)$. Hence the resulting parallel algorithm takes $O(n/K+\lg^2 n)$ time with $nK(K\geq 1)$ processors. ■

4.4 Conclusions

Recently, Shiloach and Vishkin designed a parallel algorithm for the max-flow problem which runs in $O(n^3\cdot\lg n/p)$ time using $p(1\leq p\leq n)$ processors on the WRAM[SHIL82b]. This algorithm can at best achieve the $O(n^2\lg n)$ time bound with n processors. However, they managed to derive a sequential algorithm from it which has the $O(n^3)$ time complexity. They claim that the design of parallel algorithms could provide insight into the design of sequential algorithms for the same problem. We share their feeling.

Chapter 5

PROBABILISTIC TIME, EXPECTED TIME AND O(lgn) TIME COMPLEXITIES

5.1 Introduction

In Chapter 3, it was shown that all of our algorithms run in $O(n)$ time in the worst case on the MCN. This time bound is easily seen to be optimal as routing itself takes $O(n)$ time in the worst case on that model. It was also shown that all the algorithms run in $O(\lg n)$[10] time in the worst case on the WRAM. Although Shiloach and Vishkin conjectured that it is difficult to breach this $O(\lg n)$ worst case time bound using a polynomial number of processors on the WRAM[SHIL82], no proof has been given. As for the PRAM and other more restrictive models, it was shown that the algorithms run in $O(\lg^2 n)$ time in the worst case. Although it is likely that this is a lower bound for time, no one has yet manage to prove it. It is therefore intriguing to ask: Can the $O(\lg n)$ worst case time bound be breached on the WRAM and the $O(\lg^2 n)$ worst time bound be breached on the PRAM? Recently, Reif showed that if probability error in the solution is allowed, then he could solve some of the problems in $O(\lg n)$ time with a polynomial number of processors on the PRAM and that the probability error could be eliminated by introducing *nonuniformity*[REIF82a]. More recently, Reif and Spirakis showed that some of the existing

---

[10] $L=O(\lg n)$ in the worst case.

graph algorithms do have $O(\text{loglg}n)$ expected time complexity on the WRAM and $O(\lg n\cdot\text{loglg}n)$ expected time complexity on the PRAM[REIF82b].

In this chapter, we shall show that the algorithms presented in the previous chapters could run in $O(\lg n)$ time using $|E|n^3\lg n$ processors if probability error is allowed. We shall also show that most of these algorithms have $O(\text{loglg}n)$ expected time complexity on the WRAM and $O(\lg n\cdot\text{loglg}n)$ expected time complexity on the PRAM, SIMD-CCC, OTN, OTC, CCC and PSN. Finally, we shall show that the recognition problems of split graphs and permutation graphs do have $O(\lg n)$ (deterministic) time algorithms. Reif only showed that they have $O(\lg n)$ probabilistic time algorithms[REIF82a] and no other logarithmic time algorithms were known before.

5.2 Probabilistic Time Complexity

Recently, Reif considered the possibility of breaching the $O(\lg^2 n)$ time bound for the connectivity problems and the planarity testing problem. He showed that if probability error is allowed, then the $O(\lg^2 n)$ time bound can be breached. His method is based on Aleliunas, Karp, Lipton, Lovász and Rackoff's result on random walks on connected undirected graphs[ALEL79] and Lewis and Papadimitriou's nondeterministic $O(\lg n)$ space algorithm for the UGAP problem (given a connected undirected graph $G(V,E)$ and $u,v\in V$, does there exist a path from u to v in G?)[LEWI82]. Aleliunas et

al. showed the following:

> Given any connected undirected graph $G(V,E)$, let r be a random walk in G starting from any vertex $v \in V$. r is constructed by repeated extension, randomly choosing an edge which is connected to the current front end of r and adding it to r. If r is of length $2|E|(|V|-1)$, then Prob(r visits all vertices in G)$\geq 1/2$.

Using the result of Aleliunes et al., Reif devised a probabilistic search technique to implement Lewis and Papadimitriou's UGAP algorithm in $O(\lg n)$ space.[11] Specifically, he showed that given any probability error ϵ, $0<\epsilon<1$, the UGAP problem can be solved in $O(\lg n)$ space and $n^{O(1)}$ time within error ϵ. By solving a problem within error ϵ, he means that given any problem instance ω of UGAP, if the answer to ω is *yes*, then the probability that the algorithm produces the answer *yes* is greater than or equal to $1-\epsilon$. If the answer to ω is *no*, then the probability that the algorithm produces the answer *yes* is less than ω.

Observing that deterministic PRAM's can accept within polynomial time exactly the sets that deterministic Turing machines can accept within polynomial space[GOLD78,WYLL79], Reif proceeded to show that given any probability error ϵ, $0<\epsilon<1$, the UGAP problem can be solved within error ϵ in $O(\lg n)$ time with $n^{O(1)}$ processors on the PRAM. Using this UGAP algorithm, Reif managed to implement Kruskal's greedy

---

[11] Reif's result is more formal and general. We have tailored his result here to suit our needs. Readers who are interested in his work are encouraged to consulted [REIF82a].

algorithm for the minimum spanning forest problem[HOWO79,pp.179-183] within error ϵ in $O(\lg n)$ time with $|E|\cdot p$ processors on the PRAM (p is the number of processors used by the probabilistic $O(\lg n)$ time UGAP algorithm). In [REIF82c], Reif claimed that $p=n^3\lg n$. As a result, we have:

Lemma 5.1:[REIF82a] For any probability error ϵ, $0<\epsilon<1$, there is a parallel algorithm which finds a minimum spanning forest for an undirected graph within error ϵ in $O(\lg n)$ time with $|E|n^3\lg n$ processors on the PRAM.

Using this result, it is easily shown that:

Lemma 5.2: For any probability error ϵ, $0<\epsilon<1$, there exists an $O(\lg n)$ time probabilistic parallel algorithm for finding an inverted spanning forest using $|E|n^3\lg n$ processors on the PRAM.

Proof: First, find a minimum spanning forest T for the undirected graph in $O(\lg n)$ time within probability error ϵ, $0<\epsilon<1$, using $|E|n^3\lg n$ processors[REIF82]. Then convert T into an inverted spanning forest using Lemma 3.17. ∎

Theorem 5.3: For any probability error ϵ, $0<\epsilon<1$, The class of algorithms described in Chapter 2 could run in $O(\lg n)$ time within error ϵ using $|E|n^3\lg n$ processors on the PRAM.

Proof: First note that by constructing an inverted spanning forest for an undirected graph, we can determine the

connected components of the undirected graph in $O(\lg n)$ time as follows: for every vertex v, associated v with the root of the tree in which v resides, then u, v belong to the same connected component iff u and v are associated with the same root. These roots can be identified easily if we use the array $\mathbf{F}^{+}$. The whole process clearly takes no more than $O(\lg n)$ time if $n\lceil n/\lg n\rceil$ processors are available (Theorem 2.3). As a result, by using Lemma 5.2, we can construct a directed spanning forest or determine the connected components of an undirected graph in $O(\lg n)$ time with $|E|n^3\lg n$ processors within error ϵ. Furthermore, it is easily comfirmed that all the other steps in the algorithms do not take more than $O(\lg n)$ time with $n\lceil n/\lg n\rceil$ processors. The theorem thus follows. ■

In addition to the result on random walks for connected undirected graphs, Aleliunas, Karp, Lipton, Lovász and Rackoff also gave an affirmative answer to a question from Cook concerning the existence of short n-universal sequences. An n-universal sequence is defined as follows: "Let G be a connected undirected regular graph of degree d. At each vertex v, let the edges incident with v be given the distinct labels $0,1,2,\ldots,d-1$. A sequence σ in $\{0,1,2,\ldots,d-1\}^*$ is said to traverse G from v if starting at v and following the sequence of edge labels σ, one visits all the vertices of G. σ is called an n-universal sequence if it traverses every n-vertex regular graph G with degree d starting from any vertex v." Aleliunas et al. showed that

there exists an n-universal sequence of length $O(n^3 \lg n)$.

By replacing the probabilistic choice in his probabilistic search technique with an n-universal sequence, Reif showed that the probabilistic error in his algorithm can be eliminated. However, as each n-universal sequence is good for only a particular n, the resulting algorithm becomes *nonuniform* in the sense that there is a different program for each different n. Consequently, we have:

Corollary 5.4: The set of graph theoretic problems investigated in Chapter 2 can be solved in $O(\lg n)$ time using $|E|n^3 \lg n$ processors with a nonuniform algorithm on the PRAM.

5.3 Expected Time Complexity

More recently, Reif and Spirakis showed that given a random (directed or undirected) graph, the diameter d of G has an expected length $O(\lg n)$[REIF82b]. Based on this result, they showed that some existing parallel graph algorithms, particularly, those for the graph-connectivity and minimum spanning forest, have an $O(\lg n \cdot \lg\lg n)$ expected time complexity on the PRAM and an $O(\lg\lg n)$ expected time complexity on the WRAM. Combining their results on the average length of diameters with ours stated in Chapter 3, we immediately have:

Lemma 5.5: With the exception of Algorithm Brconnect and Algorithm Biconnect, all the algorithms presented in Chapter

3 have an $O(t(n)\cdot\lg\lg n)$ expected time bound with $H(n)$ hardware resources on the *MMM*.

Proof: Since $d=O(\lg n)$ on the average[REIF82b], therefore $L=O(\lg\lg n)$. ∎

Unfortunately, Reif's result on the expected length of diameters cannot be applied to Algorithm Brconnect and Algorithm Biconnect. This is because the structures of the graphs $G_0(V_0,E_0UA_1)$ and $G''(E',E'')$ depend on the given graph $G(V,E)$ and are therefore not random graphs.

5.4 O(lgn) Time Algorithms for Split Graphs and Permutation Graphs

Split graphs and permutation graphs arise in many contexts and have received considerable attention in the past decade. The former belongs to the class of chordal graphs (triangulated graphs) which have important applications in Guassian elimination, genetic research, etc. The latter were shown to be useful in modelling and system programming like memory reallocation. The previouly known fastest sequential algorithm for identifying the split graphs takes $O(n+|E|)$ time[ROSE76,FOLD77] while that for identifying permutation graphs takes $O(n^3)$ time[EVEN72]. No parallel algorithms exist for problems of this class except Reif's $O(\lg n)$ time probabilistic parallel algorithm and $O(\lg n)$ *nonuniform* parallel algorithm[REIF82a] for the PRAM. However, his algorithm for split graphs does not generate a split if the result of the identification is positive. In

this chapter, we show that there are indeed $O(\lg n)$ (deterministic) time algorithms for the recognition problems of these two classes of graphs on the PRAM. Furthermore, the algorithm for split graphs uses $\lceil n^2/\lg n\rceil$ processors. Unfortunately, since the splitting property of a graph is not monotone, we do not know whether this processor bound is optimal. Finally, we show that these algorithms can be implemented on the *MMM* taking $O(t(n))$ time and $H(n)$ hardware resources and that the algorithms can be converted into $O(n+|E|)$ time and space optimal sequential algorithms.

5.5 Identification of Split Graphs

Lemma 5.6: $G(V,E)$ is a split graph iff G has a split $G_1(V_1,E_1)$, $G_2(V_2,E_2)$ such that G_1 is independent and G_2 is a clique.

Proof: The "if" part is obvious.

The "only if" part: If G is a split graph, then G has a split G_1, G_2 where G_2 is a complete subgraph. If G_2 is not a clique, then there exists a vertex $v \epsilon V_1$ such that $(\{v\} x V_2)$ is a subset of E. Moreover, there does not exist another $u \epsilon V_1$ for which $(\{u\} x (V_2 U \{v\}))$ is a subset of E for otherwise G_1 cannot be independent. Thus, the subgraphs $G_1'(V_1-\{v\},E_1)$, $G_2'(V_2 U\{v\},E_2 U(\{v\} x V_2))$ is a split of G in which G_2' is a clique. ■

Due to Lemma 5.6, we may, without loss of generality, assume that whenever we speak of a split G_1, G_2 of a split graph, G_1 is independent while G_2 is a clique. We will adopt

this assumption in subsequent discussion.

Apparently, if G is a split graph, then just by finding a clique G_2 in it, one should be able to conclude that G is a split graph as the remaining part G-G_2 should be independent. Unfortunately, this is not the case as is depicted in Figure 5.1.

In Figure 5.1, the graph G has three cliques. Only the one determined by the vertex set $\{a,b,d\}$ leads us to the decision that G is a split graph. It is therefore important to be able to distinguish between those cliques which would lead us to the right decision that G is a split graph (if G is indeed a split graph) and those which would not. The following lemma sheds some light on this matter.

Lemma 5.7: If $G(V,E)$ is a split graph and $G_1(V_1,E_1)$, $G_2(V_2,E_2)$ form a split,

then (i) $\deg(v) \leq |V_2|-1 \ \forall v \in V_1$;

and (ii) $\deg(w) \geq |V_2|-1 \ \forall w \in V_2$,

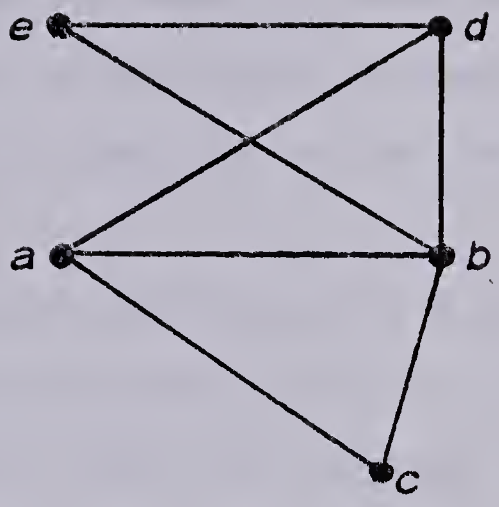
where $\deg(v)$ stands for the degree of v.

Proof: (i) Let $v \in V_1$. Then $(v,u) \notin E \ \forall u \in V_1$ because $E_1 = \emptyset$. Therefore, $\deg(v) \leq |V_2|$. But $\deg(v) = |V_2|$ implies that $G_2(V_2,E_2)$ is not a clique. Hence, $\deg(v) \leq |V_2|-1$.

(ii) Immediate from the definition of complete graphs. ■

Corollary 5.8: Let $G(V,E)$ be a split graph. For any $u \in V_1$, $v \in V_2$, $\deg(u) \leq \deg(v)$.

Proof: Immediate from Lemma 5.7. ■



{*a*,*b*,*c*},{*a*,*b*,*d*}, and {*b*,*d*,*e*} are cliques.

Only {{*a*,*b*,*d*},{*c*,*e*}} induces a split.
{{*a*,*b*,*c*},{*d*,*e*}} and {{*b*,*d*,*e*},{*a*,*c*}} don't.

Figure 5.1

Corollary 5.8 indicates that if we sort the vertex set V by degree of vertex in descending order, then those vertices in V_2 will precede those in V_1 in the sorted sequence with an intermixed region inbetween which contains the set of all vertices in V_1 and V_2 having the same degree $|V_2|-1$. Therefore to identify the vertex set of the clique G_2, we have to be able to identify those vertices of V_2 in the intermixed region of the sorted sequence. Fortunately, this is not a difficult task due to the following lemma.

Lemma 5.9: Let $G_1(V_1,E_1)$, $G_2(V_2,E_2)$ be a split of a split graph $G(V,E)$. Let $C_1=\{u\in V_1 \mid \deg(u)=|V_2|-1\}$ and $C_2=\{v\in V_2 \mid \deg(v)=|V_2|-1\}$; then for any $u\in C_1$ and any $v\in C_2$, $G_1'(V_1-\{u\}\cup\{v\},E_1)$ and $G_2'(V_2-\{v\}\cup\{u\},E_2\cup(\{u\}\times(V_2-\{v\}))-(V_2\times\{v\}))$ is also a split of G.

Proof: Since $\deg(v)=|V_2|-1$ and G_2 is a clique, $(u,v)\notin E$. Furthermore, as $\deg(u)=|V_2|-1$, and G_1 is independent, $G_2'(V_2-\{v\}\cup\{u\},E_2\cup(\{u\}\times(V_2-\{v\}))-(V_2\times\{v\}))$ must be a clique. Since $\deg(v)=|V_2|-1$ and $v\in V_2$, $\therefore (w,v)\notin E$, $\forall w\in V_1$. This implies that $G_1'(V_1-\{u\}\cup\{v\},E_1)$ is independent. Hence, G_1', G_2' is a split of G. ∎

The above lemma implies that if we sort the vertex set of a split graph G by degree of vertex in descending order, then the first k vertices and the remaining $n-k$ vertices in the sorted sequence always constitute the vertex sets of a split of G where $k=\max\{i \mid \deg(v_i)\geq i-1\}$. v_i is the ith vertex

in the sorted sequence.

Hence we have the following characterization theorem for split graphs.

Theorem 5.10: Let $G(V,E)$ be an undirected graph and v_1, v_2, v_3, ... , v_n be the sequence of vertices of G sorted by degree of vertex in descending order.
G is a split graph iff $\{v_1, v_2, v_3, \ldots, v_k\}$ induce a clique in G while $\{v_{k+1}, v_{k+2}, \ldots, v_n\}$ induce an indendent subgraph of G where k is defined as above.

Proof: By Lemmas 5.7, 5.9 and the definition of split graph. ■

```
Algorithm : Split
(* This algorithm examines if an undirected graph is a split
   graph and produces a split if it is.
   deg($v$) stands for degree of $v$ *)

1. Compute deg($v$) $\forall v \in V$. Sort $V$ by degree of vertex in
   descending order.
2. Find $k$ such that $k=\max\{i \mid \deg(v_i)\geq i-1 \text{ and } \deg(v_{i+1})\leq i-1\}$;
3. Let $V_1=\{v_{k+1},v_{k+2},\ldots,v_n\}$; $V_2=\{v_1,v_2,\ldots,v_k\}$.
   Check if $V_2$ form a clique in $G$. If not, then $G$ is not a
   split graph.
4. Check if $V_1$ induce an independent set in $G$. If no, then $G$
   is not a split graph.
5. Declare $G$ is a split graph and $G_1(V_1,E_1)$, $G_2(V_2,E_2)$ is a
   split. ■
```

Theorem 5.11: Algorithm Split correctly identifies a split graph.

Proof: Given any graph $G(V,E)$, If G is a split graph, then

by Theorem 5.10, Algorithm Split correctly identify G as a split graph. If G is not a split graph, then either Step 3 detects that $G_2(V_2,E_2)$ is not a clique or Step 4 detects that $G_1(V_1,E_1)$ is not independent. In either case, G is identified not to be a split graph. ▪

Theorem 5.12: Algorithm Split runs in $O(n/\mathrm{K}+\lg n)$ time with $n\mathrm{K}(\mathrm{K}\geq \lg n)$ processors on the PRAM.

Proof: Given an adjacency matrix M of G, Step 1 takes $O(n/\mathrm{K}+\lg n)$ time with $n\mathrm{K}(\mathrm{K}>0)$ processors to compute $\deg(v)$ $\forall v\in V$ (Lemma 2.2) and $O(\lg n)$ time with $n\lg n$ processors to sort the vertices by degree of vertex[BORO82]. Step 2 takes $O(1)$ time with n processors. Step 3 takes $O((k-1)/\mathrm{K}+\lg n)$ time with $n\mathrm{K}(\mathrm{K}>0)$ processors. Step 4 takes $O((n-k-1)/\mathrm{K}+\lg n)$ time with $n\mathrm{K}(\mathrm{K}>0)$ processors. Hence, Algorithm Split runs in $O(n/\mathrm{K}+\lg n)$ time with $n\mathrm{K}(\mathrm{K}\geq \lg n)$ processors. ▪

Corollary 5.13: Identifying a split graph can be done in $O(\lg n)$ time with $n\lceil n/\lg n\rceil$ processors for $\lceil n/\lg n\rceil\geq \lg n$.

Now, we shall implement Algorithm Split on the *MMM*. To ease the task of explanation, we shall assume that the degrees of the vertices of $G(V,E)$ are all distinct. Generalizing our result to arbitrary case is straightforward.

Theorem 5.14: Algorithm Split runs in $O(t(n))$ time with $\mathrm{H}(n)$

hardware resources on the *MMM*.

Proof: Let M be the adjacency matrix. In step 1, deg(v) is computed by:

$deg[u,v]:=\Sigma_{k=1}^{n} M[u,k];$

(clearly deg(u)=$deg[u,u]=deg[u,v]\ \forall u,v \in V$).

Order the vertices by degree of vertex as follows:

(Broadcast columnwise:) $deg'[w,v]:=deg[v,v]\ \forall v,w \in V;$

$rank[u,v]:=\Sigma_{k=1}^{n}(deg[u,k] \le deg'[u,k]);$

(note that rank(u)=$rank[u,u]=rank[u,v]$, and rank(u) is the position of u in the sorted sequence).

In step 2, k is determined as follows:

(Erase the rank of those u whose rank does not satisfy the condition : deg(u)≥rank(u)-1.)

```
rank[u,u]:=if (deg[u,u]≥rank[u,u]-1)
               then rank[u,u]
               else 0;
```

$great[u,u]:=\neg(\bigvee_{k}(rank[u,k]<rank[k,u]));$

(note that $great[u,u]=1$ iff u is the vertex whose rank is the k).

In step 3, the set V is partitioned as follows:

(Broadcast columnwise:) $great'[w,v]:=great[v,v];$

(Broadcast columnwise:) $rank'[w,v]:=rank[v,v];$

$Split[u,v]:=(rank[u,v] \le rank'[u,v]) \wedge great'[u,v];$

(Broadcast the nonzero Split rowwise. Note that there is at most one nonzero Split value on each rowwise):

$Split[u,u]:=\Sigma_{k=1}^{n} Split[u,k];$

as a result, $V_1=\{u \mid Split[u,u]=0\}$ and

$V_2=\{u|Split[u,u]=1\}$.

Steps 4 and 5 can be combined and tested together as below:

(Broadcast rowwise:) $Split[v,w]:=Split[v,v]$:

(Broadcast columnwise:) $Split'[w,v]:=Split[v,v]$;

$Flag[u,v]:=(\neg Split[u,v]\wedge Split'[u,v])$

$\vee\ (Split[u,v]\wedge\neg Split[u,v])$

$\vee\ (Split[u,v]\wedge Split'[u,v]\wedge M[u,v])$

$\vee\ (\neg Split[u,v]\wedge\neg Split[u,v]\wedge\neg M[u,v])$

The above statement should be interpreted as $Flag[u,v]=1$ iff $u\epsilon V_1$ and $v\epsilon V_2$ or $u\epsilon V_2$ and $v\epsilon V_1$ or $(u,v)\epsilon E$ if $u,v\epsilon V_2$ or $(u,v)\notin E$ if $u,v\epsilon V_1$. Hence, G is a split graph iff $Flag[u,v]=1$ $\forall u,v\epsilon V\mathrm{x}V$. Therefore, after computing:

$Flag[u,v]:=\wedge_k(Flag[u,k]\wedge Flag[k,v])$

twice, $Flag[1,1]=1$ iff G is a split graph. ■

Theorem 5.15: Algorithm Split runs on a sequential computer in $O(n+|E|)$ time and space.

Proof: In Step 1, we use bucket sort[AHO74, Section 3.2] to sort the vertices in V. This takes linear time and space. Step 2 takes $O(n)$ time. Steps 3 and 4 takes $O(n+|E|)$ time. Moreover, $O(n+|E|)$ space is sufficient if we use an adjacency list to represent the graph.

From Lemmas 3.1, 3.2 and 3.3, we have the indicated time and hardware resource complexities. ■

5.6 Identification of Permutation Graphs

Our algorithm is based on the following characterization theorem due to Even, Pnueli and Lempel.

Theorem 5.16: An undirected graph $G(V,E)$ is a permutation graph iff both $G^{+}(V,E^{+})$ and $G^{-}(V,E^{-})$ are transitive, where G^{+} and G^{-} are directed graphs induced from G such that $E^{+}=\{<i,j>|\, i<j \text{ and } (i,j)\in E\}$ and $E^{-}=\{<i,j>|\, i>j \text{ and } (i,j)\in V\text{x}V-(E\cup\{(v,v)|v\in V\})\}$.

Proof: [EVEN72]. ∎

Even, Pnueli and Lempel also showed how to determine the permutation of G if G is a permutation graph. Specifically, they formed $G^{\circ}=G^{+}\cup G^{-}$ and showed that G° is a cycle-free directed graph whose underlying graph is complete. Therefore, G° must have a sink s_1, namely, a vertex which has no outgoing edge. They removed s_1 from G° and showed that the resulting graph remains cycle-free and its underlying graph is also complete. A sink s_2 in this graph therefore exists. By repeating this process, they ended up with a sequence of sinks $s_1,s_2,...,s_n$. They showed that the permutation P such that $P(i)=s_i$, $1\leq i\leq n$ is a permutation of G. To determine P efficiently in parallel, we restate P as:

$$P=\{<i,P(i)>|\, \mathit{out\text{-}degree}(P(i))=i-1 \text{ in } G^{\circ},\ 1\leq i\leq n\}.$$

Based on there results, we immediately have:

Theorem 5.17: Identifying a permutation graph and

determining its permutation can be done in $O(n/K+\lg n)$ time with $n^2K (K\geq 1)$ processors on the PRAM.

Proof: Constructing G^+ and G^- from G takes $O(n/K+\lg K)$ time with $nK (K\geq 1)$ processors(Lemma 2.2). Testing if both G^+ and G^- are transitive takes $O(n/K+\lg n)$ time with $n^2K (K\geq 1)$ processors(Lemma 2.2). Moreover, if G is a permutation graph, then determining the permutation P of G takes $O(n/K+\lg K)$ time with $nK (K\geq 1)$ processors as this is the time and processor complexities one needs to compute *out-degree*(v), for all v in G°. ■

Corollary 5.18: Identifying a permutation graph and determining its permutation can be done in $O(\lg n)$ time with $n^3/\lg n$ processors on the PRAM.

Theorem 5.19: Identifying a permutation graph and determining its permutation can be done in $O(t(n))$ time with $H(n)$ hardware resources on the *MMM*.

Proof: Trivial. ■

5.7 Conclusions

Breaching the $O(\lg^2 n)$ time bound for graph theoretic problems on the PRAM is one of the main concern in algorithm design. It seems to be a difficult task, and in fact some have conjectured that it is impossible[KUCE82].

In this chapter, although we do not manage to develop a general technique to surpass the $O(\lg^2 n)$ time bound, we do

show that most of our algorithms have probabilistic time complexity and expected time bound below $O(\lg^2 n)$ and that the recognition problems for split graphs and permutation graphs have $O(\lg n)$ optimal time complexity. (Note that although the lowest common ancestor algorithm has $O(\lg n)$ time complexity, the graphs it deals with are directed trees which is only a subset of all graphs). We feel that our success in finding $O(\lg n)$ time algorithms for split graphs and permutation graphs is due to the particular characteristic theorems for these graphs. These characteristic theorems allow us to process the graph locally at each vertex without having to perform a graph search to collect global information. This is reflected in the algorithms by the fact that no construction of an inverted spanning forest is necessary. The process of collecting global information is a time-consuming process and is the main cause of the $O(\lg^2 n)$ time complexity. As a result, we believe that one way to breach the $O(\lg^2 n)$ time bound for graph theoretic problems is to develop characteristic theorems which allow us to get global information without performing a graph search. However, discovering such characteristic theorems seems to be very difficult in general.

Finally, as with Chapter 2, we remark that the lower bound for the number of processors used in identifying split graphs and permutation graphs can be reduced to $n\mathrm{K}(\mathrm{K}>0)$ rather than $n\mathrm{K}(\mathrm{K}\geq 1)$.

Chapter 6

CONCLUSIONS

We have presented algorithms for the class of graph theoretic problems listed in the introduction to this thesis. These algorithms achieve the conjectured lower bound for the worst case time complexity on many of the existing models. The number of processors they require is optimal in most cases for the PRAM. Furthermore, they have good expected time complexities and have $O(\lg n)$ probabilistic time on the PRAM. In most cases, the results obtained provide new upper bounds for the problems. Hence, we believe that the goals of 'portability' and 'efficiency' have both been achieved.

The concept of 'portability' is not new in the discipline of computer science, and is certainly an important one. Surprisingly, such an important concept has not received much attention in the design of efficient algorithms for parallel computer models. Although some portable algorithms have appeared in the literature, their portability was made possible by the simplicity of the problems, and not the design of the algorithms. The first work (possibly the only work) emphasizing the concept of portable algorithms was an unpublished manuscript of Miller and Stout for the graph-connectivity problem[MILL82]. However, the class of computers on which their algorithm works efficiently is relatively small. In our opinion, the *MMM* proposed in this thesis serves as a good model for

designing efficient portable algorithms. There are several reasons for this: firstly, it has a great deal of generality. In this thesis it has been shown that it includes most of the well-known existing models. Therefore, any algorithm which runs on the *MMM* will automatically run on all those models. Secondly, it has been shown that many operations can be carried out in $O(t(n))$ time with $H(n)$ hardware resources on the *MMM*. These include the prototype operations like sorting, labelling the vertices, computing partial sums, finding the maximum and minimum, etc. We have seen that $t(n)$ is also the lower time bound for graph theoretic problems on many of the existing computer models. As a result, we may employ any of these prototype operations freely in designing graph algorithms on the *MMM* as the time they consume is always within the optimal time bound. In other words, we stand a good chance of getting optimal graph algorithms on the *MMM*. Thirdly, matrix multiplication is a basic yet important operation. Its central role in many scientific applications is widely recognized. Any computer model whose design is unsuitable for matrix multiplication will be of limited usefulness. For this reason, it may be justified to say that any general purpose computer model is an *MMM*. Finally, due to the uniform nature of ordinary matrix multiplication, the *MMM* should be easily constructed (note that the processors need not have expensive multiplication capability). The number of processors required is also reasonable, since otherwise it may not be

possible to realize matrix multiplication efficiently on parallel computer models.

We feel that the directed spanning forest problem deserves more attention. The importance of this problem seems to have been overlooked after the search for efficient (sublinear time) parallel depth-first search algorithm was unsuccessful. In fact, the importance of this problem is easy to appreciate as the directed spanning forest provides a framework upon which global information can be organized and transferred from vertex to vertex within the graph. The success of the depth-first search technique (which creates a directed spanning forest) in designing optimal algorithms for the sequential RAM gives strong support to this view. The fact that the directed spanning forest for the PRAM and the directed BFS spanning forest for the *MMM* serve as the backbone of all of our algorithms provides further evidence. Moreover, in the course of developing our algorithms, we observed that the execution times of our algorithms are dominated by the directed spanning forest algorithm. This is because with the exception of the steps for finding a directed spanning forest and for determining the connected components of an undirected graph G, we have ensured that all the steps in our algorithms run in optimal time. But we have shown in Theorem 5.3 that the connected component problem can be reduced to a directed spanning forest problem. Therefore the optimality of our algorithms depends

on our ability in developing an optimal directed spanning forest algorithm. In other words, we may reduce the problem of finding an optimal time algorithm for any of the graph theoretic problems investigated in this thesis to that of finding a directed spanning forest of an undirected graph. This may explain why, in Chapter 5 whenever there is an improvement in the directed spanning forest algorithm, there is automatically an improvement in all the other algorithms.

In view of the importance of the directed spanning forest problem, we summarize our results on this problem and propose several related open problems below.

1. A directed spanning forest can be found in $O(\mathrm{lg}^2 n)$ time with $n\lceil n/\mathrm{lg}^2 n\rceil$ processors. This result is optimal for dense graphs with respect to the time-processor product. (Chapter 2)
2. A directed spanning forest can be found in $O(\mathrm{lg} n)$ probabilistic time with $|E|n^3\mathrm{lg} n$ processors on the PRAM. (Chapter 5)
3. A directed BFS spanning forest can be found in $O(\mathrm{lg} n\cdot\mathrm{lglg} n)$ expected time on the PRAM and in $O(\mathrm{lglg} n)$ expected time on the WRAM with n^3 processors. (Chapter 3)
4. A directed BFS spanning forest can be found in $O(\mathrm{lg} n\cdot\mathrm{lg} d)$ time with $\mathrm{H}(n)$ hardware resources on the *MMM* where d is the diameter of the given graph.

Besides the direct way of finding a directed spanning forest, two alternative indirect ways have been used in this thesis. The first is to find a minimum spanning forest for the given graph (note that the minimum spanning forest does not convey global information efficiently) and then convert it into a directed forest by constructing a directed BFS spanning forest in it. This technique was described in Lemma 3.17 and was employed in Chapter 5. The second way is to use the all-pair shortest path algorithm. This technique has been used in Chapter 3 to produce a directed BFS spanning forest.

The following are open problems:

1. Can a directed spanning forest be found in $O(\lg n)$ time with $n\lceil n/\lg n\rceil$ or even $\lceil |E|/\lg n\rceil$ processors on the PRAM? Note that solving this problem implies solving all the graph theoretic problems investigated in this thesis in optimal time using an optimal number of processors on the PRAM.
2. Can the number of processors used by the $O(\lg n)$ time probabilistic algorithms be reduced?
3. Can the expected time complexities be improved or the number of processors used be reduced?
4. Can the time complexity be improved on the *MMM*?

Since the majority of the problems investigated in this thesis are related to the connectivity property of graphs,

it is natural to ask if the k-connectivity($K\geq 3$) problems can be solved efficiently. The best previously known parallel algorithm for testing if a graph is k-connected($k\geq 3$) on the PRAM takes $O(\lg^2 n \lg k)$ time with $O(n^{k+1})$ processors[GOLD77] or $O(\lg n)$ probabilistic time with $n^{o(1)}$ processors[REIF82a]. No algorithms were known for other parallel computer models. Using the results obtained in the previous chapters for the biconnectivity problem and the following lemma, it is easily shown that testing if an undirected graph is k-connected($k\geq 3$) can be done in $O(\lg^2 n)$ time with $n^{k-1}\lceil n/\lg^2 n\rceil$ processors or in $O(\lg n)$ probabilistic time with $|E|n^{k+1}\lg n$ processors on the PRAM. The worst case and the expected time complexities for the *MMM* can be similarly derived.

Lemma 6.1: An undirected graph G is k-connected($k\geq 3$) iff $\forall (v_1,v_2,\ldots,v_{k-2})\epsilon V^{k-2}$, $G[v_1,v_2,\ldots,v_{k-2}]$ is biconnected, where $G[v_1,v_2,\ldots,v_{k-2}]$ is obtained from G by removing the k-2 vertices v_1, v_2, ..., v_{k-2} and all the edges incident with these vertices from G.

Proof: Trivial. ∎

Finding the k-connected components for $k\geq 4$ is of no practical interest. However, for k=3, the problem is closely related to the planar graph problem which has application in electrical engineering. The previous best algorithm for the 3-connected components takes $O(\lg^2 n)$ time with n^4 processors on the PRAM[JAJA82]. No algorithms for other parallel

computer models were known. Using Lemma 6.1, it is easily shown that we can improve Ja'Ja' and Simon's 3-connected components algorithm by reducing the number of processors used by a factor of $\lg^2 n$.

Despite the fact that our results obtained here for k-connectivity problem ($k \geq 3$) are improvements over the previous results, we do not regard them as achievements because the method proposed by Lemma 6.1 is essentially a brute force method, let alone the fact that the results are not optimal (In fact, all the previous results stated above are to a great extent, brute force methods).

At this point, it is interesting to review the results for these problems obtained on the sequential RAM. The sequential algorithms for finding the connected components, the biconnected components and the triconnected components all rely on the depth-first search technique and run in optimal time and space[TARJ72,HOPC73]. Since we have developed an optimal (w.r.t. time-processor product) directed spanning forest algorithm in Chapter 2, based on which an optimal biconnected component algorithm was developed, and we have shown in Chapter 4 that the biconnected component algorithm gives rise to an optimal sequential algorithm which is a generalization of the previous optimal sequential algorithm. It is therefore intriguing to ask: Using the optimal directed spanning forest algorithm, can we develop an optimal parallel algorithm for the triconnected component problem which gives

rise to an optimal sequential algorithm which is a generalization of the existing optimal sequential algorithm on the PRAM? We leave this as an open problem.

Bibliography

[AHO74] Aho,A., J.E.Hopcroft, J.Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading, MA, 1974.

[ALEL79] Aleliunas,R., R.Karp, R.Lipton, L.Lovász, C.Rackoff, "Random walks, universal traversal sequences, and the complexity of maze problems," *Proc. 20th Annual* IEEE *Symposium on Foundations of Computer Science*, San Juan, Puerto Rico, October 1979, pp.218-223.

[ARJO75] Arjomandi,E., *A Study of Parallelism in Graph Theory*, Ph.D thesis, Dept of Computer Science, U of Toronto, Toronto, December 1975.

[ATAL82] Atallah,M., S.Kosaraju, "Graph problems on a mesh-connected processor array," *Proc. 14th Annual* ACM *Symposium on Theory of Computing*, San Francisco, CA, May 1982, pp.345-353.

[ATAL83] Atallah,M., "Finding Euler tours in parallel," 1983, submitted to *JCSS*.

[AWER83] Awerbuch,B., Y.Shiloach, "New connectivity and MSF algorithms for Ultracomputer and PRAM," to appear in *Proc.* IEEE *International Conference on Parallel Processing*, Bellaire, Michigan, August 1983.

[BATC68] Batcher,K.E., "Sorting network and their applications," *Proc. AFIPS Spring Joint Computer Conf.*, vol.32, Atlantic City, April-May 1968, pp.307-314.

[BORO82] Borodin,A., J.E.Hopcropt, "Routing, merging and sorting on parallel models of computation," *Proc. 14th Annual* ACM *Symposium on Theory of Computing*, San Francisco, CA, April 1982, pp.338-344.

[BREN74] Brent,R.P., "The parallel evaluation of general arithmetic expressions," **J.ACM** vol.21, No.2, April 1974, pp.201-206.

[CANN69] Cannon,L.E., *A cellular computer to implement the Kalman Filter algorithm*, Ph.D.Thesis, Montana State University, 1969.

[CHAN76] Chandra,A.K., "Maximal parallelism in matrix multiplication," *IBM Research Report*, RC 6193, Thomas J.Watson Research Center, 1976.

[CHIN81] Chin,F.Y., J.Lam, I-Ngo Chen, "Optimal parallel algorithm for the connected component problem," *Proc.* **IEEE** *International Conference on Parallel Processing*, Bellaire, Michigan, August 1981, pp.170-175.

[CHIN82] Chin,F.Y., J.Lam, I-Ngo Chen, "Efficient parallel algorithms for some graph problems," **CACM** vol.25, No.9, September 1982, pp.659-665.

[CHRI73] Christopher,T.W., "An implementation of Warshall's algorithm for transitive closure on a cellular computer," Rep.No.36, Inst. of Comp. Res. University of Chicago, February 1973.

[CHVA73] Chvátal,V., P.L.Hammer, "Set-packing problems and thresold graphs," U of Waterloo, CORR 73-21, August 1973.

[DEKE81] Dekel,E., D.Nassimi, S.Sahni, "Parallel matrix and graph algorithms," **SIAM** *J.Computing*, vol.10, No.4, November 1981, pp.657-675.

[ECKS77a] Eckstein,D.M., *Parallel Graph Processing Using Depth-first Search and Breath-first Search*, Ph.D. thesis, Dept. of Computer Science, U of Iowa, Iowa City, Iowa, 1977.

[ECKS77b] Eckstein,D.M., D.A.Alton, "Parallel graph processing using depth-first search," *Conf. on Theoretic Comp. Sci.*, U. of Waterloo,1977, pp.21-29.

[ESWA76] Eswaran,K.P., R.Tarjan, "Augmentation problems," **SIAM** *J.Computing* vol.5, No.4, December 1976, pp.653-665.

[EVEN72] Even,S., A.Pnueli, A.Lempel, "Permutation graphs and transitive graphs," **J.ACM**, vol.19, No.3, July 1972, pp.400-410.

[EVEN75] Even,S., "An algorithm for determining whether the connectivity of a graph is at least k," **SIAM** *J.Computing*, vol.4, No.3, September 1975, pp.393-396.

[EVEN79] Even,S., *Graph Algorithms*, Computer Science Press, Potomac, Maryland, 1979.

[FLYN66] Flynn,M.J., "Very high speed computing systems," *Proc.***IEEE**, vol.54, December 1966, pp.1901-1909.

[FLYN72] Flynn,M.J., "Some computer organizations and their effectiveness," **IEEE** *Trans. on Computers*, vol.C-21, No.9, September 1972, pp.948-960.

[FOLD76] Foldes,S., P.L.Hammer, "On a class of

matroid-producting graphs," U of Waterloo, CORR 76-6, March 1976.

[FOLD77] Foldes,S., P.L.Hammer, "Split graphs," *Proc. 8th Sourtheastern Conf. Combinatorics and Graph Theory and Computing*, Baton Rouge, Louisiana, February-March 1977, pp.311-315.

[FORT78] Fortune,S., J.Wyllie, "Parallelism in random access machines," *Proc. 10th Annual* **ACM** *Symposium on Theory of Computing*, San Diego, CA, December 1978, pp.114-118.

[GOLD77] Goldschlager,C.M., *Synchronous Parallel Computation*, Ph.D. thesis, TR-114, Dept. of Computer Science, U of Toronto, Toronto, December 1977.

[GOLD78] Goldschlager,C.M., "A unified approach to models of synchronous parallel machines," *Proc. 10th Annual* **ACM** *Symposium on the Theory of Computing*, San Diego, CA, May 1978, pp.89-94.

[GUIB79] Guibas,L.J., H.T.Kung, C.D.Thompson, "Direct VLSI implementation of combinatorial algorithms," *Proceeding 1979 Caltech Conf. on VLSI*, pp.509-525.

[HARA69] Harary,F., *Graph Theory*, Addison-Wesley, Reading, MA. 1969.

[HARE80] Harel,D., "A linear time algorithm for the lowest common ancestors problem," *Proc. 21th Annual* **IEEE** *Symposium on the Foundations of Computer Science*, Syracuse, N.Y., October 1980, pp.308-319.

[HIRS76] Hirschberg,D.C., "Parallel algorithms for the transitive closure and the connected component problems," *Proc. 8th Annual* **ACM** *Symposium on Theory of Computing*, Hershay, PA, May 1976, pp.55-57.

[HIRS78] Hirschberg,D.S., "Fast parallel sorting algorithms," **CACM**, vol.21, No.8, August 1978, pp.657-661.

[HIRS79] Hirschberg,D.S., A.K.Chandra, D.V.Sarwate, "Computing connected components on parallel computers," **CACM**, Vol.22, No.8, August 1979, pp.461-464.

[HOPC74] Hopcroft,J.E. and R.Tarjan, "Efficient planarity testing," **J.ACM**, vol.2, No.4, October 1974, pp.549-568.

[HOPC73] Hopcroft,J.E., R.Tarjan, "Dividing a graph into triconnected components," **SIAM** *J.Computing* vol.2, No.3, September 1973, pp.135-158.

[HORO79] Horowitz,E. S.Sahni, *Fundamental of Computer*

Algorithms, Computer Science Press, Potomac, MD, 1979.

[JAJA78] Ja'Ja',J., "Graph connectivity problems on parallel computers," *Technical Report* CS-78-05, Dept. of Computer Science, Pennsylvania State University, University Park, February 1978.

[JAJA82] Ja'Ja',J., J.Simon, "Parallel algorithms in graph theory:planarity testing," SIAM *J. Computing*, vol.11, No.2, May 1982, pp.314-328.

[KIRK74] Kirkpatrick,D., "Determining graph properties from matrix representations," *Proc. 6th Annual* ACM *Symposium on Theory of Computing*, Seattle, Washington State, April 1974, pp.84-90.

[KNUT73] Knuth,D., *The Art of Computer Programming*, vol.1., Addison Welsey, Reading, MA, 2nd ed., 1973.

[KOGG73] Kogge,P., H.Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," IEEE *Trans. on Computers*, vol.C-22, No.8, August 1973, pp.786-792.

[KRUS82] Kruskal,C.P., "Algorithms for Replace-add-based paracomputers," *Proc.* IEEE *International Conference on Parallel Processing*, August 1982.

[KUCE82] Kućera,L., "Parallel computation and conflicts in memory access," *Info. Processing Letters*, vol.14, No.2, April,1980, pp.93-96.

[KUNG67] Kung,H.T., "Synchronous and asynchronous parallel algorithms for multiprocessors," in *Algorithms and Complexity*, edited by J.F.Traub, Academic Press, N.Y., 1967, pp.153-200.

[KUNG80] Kung,H.T., "The structure of parallel algorithms," in *Advances in Computers*, Vol.19, Academic Press, New York, 1980, pp.65-112.

[LEWI82] Lewis,H.R., C.H.Papadimitriou, "Symmetric space-bounded computation," *Theoretical Computer Science* 19, 1982, pp.161-187.

[MILL82] Miller,R., Q.F.Stout, "A portable component labeling algorithm," (Perliminary Version) SUNY Binghamton N.Y., May 1982, unpublished manuscript.

[MUNR73] Munro,I., M.Paterson, "Optimal algorithms for parallel polynomial evaluation," *JCSS*, vol.7, No.2, April 1973, pp.189-198.

[NASS80] Nassimi,D., S.Sahni, "Finding connected components

and connected ones on a Mesh-connected parallel computer," **SIAM** *J. Comput.*, vol.9, No.4, November 1980, pp.744-757.

[NATH81] Nath,D., S.N.Maheshwari, P.C.P.Bhatt, "Efficient VLSI networks for parallel processing based on orthogonal trees," **IEEE** *Trans. on Computers*, vol.C-32, No.6, June 1983, pp.569-581.

[NATH82] Nath,D., S.N.Maheshwari, "Parallel algorithms for the connected components and minimal spanning problems," *Info. Processing Letters*, vol.14, No.1, March 1982, pp.7-11.

[PELE75] Peled,U.N., *Regular boolean functions and their polytropes*, Chapter III, Ph.D.thesis, Dept of Combinatorics and Optimization, U of Waterloo, Ontario, 1975.

[PREP78] Preperata,F.P., "New parallel-sorting scheme," **IEEE** *Trans. on Computers*, vol.C-27, No.7, July 1978, pp.669-673.

[PREP81] Preparata,F.P., J.Vuillemin, "The cube-connected cycles : A versatile network for parallel computation," **CACM**, vol.24, No.5, May 1981, pp.300-309.

[REGH78] Reghbati,E., D.G.Corneil, "Parallel computations in graph theory," **SIAM** *J.Computing*, Vol.7, No.2, May 1978, pp.230-237.

[REIF82a] Reif,J., "Symmetric complementation," *Proc. 14th Annual* **ACM** *Symposium on Theory of Computing*, San Francisco, CA, April 1982, pp.201-214.

[REIF82b] Reif,J., P.Spirakis, "The expected time complexity of parallel graph and digraph algorithms," *Technical Report* TR-11-82, Harvard University, Cambridge, MA, U.S.A., October 1982.

[REIF82c] Reif,J., "On synchronous parallel computations with independent probability choice," *Technical Report* TR-30-81, Harvard University, Cambridge, MA, U.S.A., September 1982.

[REIN77] Reingold E.M.,J.Nievergelt,N.Deo, *Combinatorial Algorithms: Theory and Pratice*, Prentice Hall, Englewood Cliffs, N.J., 1977.

[RIVE76] Rivest,R.L., J.Vuillemin, "On recognizing graph properties from adjacency matrices," *Theoretical Computer Science*,3 (1976)pp.371-384.

[ROSE70] Rose,D.J., "Triangulated graphs and elimination

process," *Journal of Math Anal & Appli* vol.32, No.3, December 1970, pp.597-609.

[ROSE76] Rose D.J., R.E.Tarjan, G.S.Lueker, "Algorithm aspects of vertex elimination on graphs," **SIAM** *J.Computing*, vol.5, No.2, June 1976, pp.266-283.

[SAVA77] Savage,C.D., *Parallel Algorithms for Graph Theoretic Problems*, Ph.D. Dissertation,R-784, Dept. of Math., U. of Illinois, Urbana, 1977.

[SAVA81] Savage,C.D., J.Ja'Ja', "Fast, efficient parallel algorithms for some graph problems," **SIAM** *J.Computing*, Vol.10, No.4, November 1981, pp.682-691.

[SCHW80] Schwartz,J.T., "Ultracomputers," **ACM** *Trans. on Prog. Lang. and Systems*, Vol.2, No.4, October 1980, pp.484-512.

[SIEG77] Siegel,H., "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," **IEEE** *Trans. on Computers*, vol.C-26, No.2, February 1977, pp.153-161.

[SIEG79] Siegel,H., "A model of SIMD machines and a comparison of various interconnection networks," **IEEE** *Trans. on Computers*, vol.C-28, No.12, December 1979, pp.907-917.

[SHIL81] Shiloach,Y., U.Vishkin, "Finding the maximum, merging and sorting in a parallel computation model," *Journal of Algorithms*, vol.2, No.1, March 1981, pp.88-102.

[SHIL82a] Shiloach,Y., U.Vishkin, "An $O(\log n)$ parallel connectivity algorithm," *Journal of Algorithms*, vol.3, No.1, March 1982, pp.57-67.

[SHIL82b] Shiloach,Y., U.Vishkin, "An $O(n^2 \log_2 n)$ parallel MAX-FLOW algorithm," *Journal of Algorithms*, vol.3, No.2, June 1982, pp.128-148.

[STON71] Stone,H.S., "Parallel processing with the perfect shuffle," **IEEE** *Trans. on Computers*, vol.C-20, No.2, February 1971, pp.153-161.

[STON73] Stone,H., "Problems of parallel computation" p.4 and p.12 in *Complexity of Sequential and Parallel Numerical Algorithms* , edited by J.F.Traub, Academic Press, N.Y., 1973.

[STON80] Stone,H., "Parallel computers," in *Introduction to Computer Architecture*, edited by H.S.Stone, SRA, Chicago Ill., 1980, pp.318-374.

[STRA69] Strassen,V., "Gaussian elimination is not optimal," Num. Math., 13, 1969, pp.345-356.

[TARJ72] Tarjan,R., "depth-First search and linear graph algorithms," **SIAM** *J.Computing* vol.1, No.2, June 1972, pp.146-160.

[TARJ74] Tarjan,R., "A note on finding the bridges of a graph," *Info. Processing Letters* vol.2, No.2, 1974, pp.160-161.

[TARJ82] Tarjan,R. "Graph partitions defined by simple cycles," March 1982, submitted to *J. Graph Theory*.

[TSIN82a] Tsin,Y.H. and F.Chin, "Efficient parallel algorithms for a class of graph theoretic problems," to appear in **SIAM**, *Journal on Computing*.

[TSIN82b] Tsin,Y.H., "A generalization of Tarjan's depth first search algorithm for the biconnectivity problem," *14th Southeastern Conference on Combinatorics, Graph Theory, Computing*, Boca Raton, Florida, February 1983.

[TSIN83a] Tsin,Y.H., "Bridge-connectivity and biconnectivity algorithms for parallel computer models," to appear in *Proc.* **IEEE** *International Conference on Parallel Processing*, Bellaire, Michigan, August 1983.

[TSIN83b] Tsin,Y.H., F.Chin, "A general program scheme for finding bridges," to appear in *Info. Processing Letters*.

[VALI75] Valiant,L.G., "Parallelism in comparison problems," **SIAM** *J.Computing*, vol.4, No.3, September 1975, pp.348-355.

[VANS80] Van Scoy,F.C., "The parallel Recognition of Classes of Graphs," **IEEE** *Trans. on Computers*, vol.C-29, No.7, July 1980, pp.563-572.

[VISH81a] Vishkin,U., *Synchronized parallel computation*, D.Sc. thesis, Computer Science Department, Technion, Haifa, Israel, February 1981.

[VISH81b] Vishkin,U., "Implementation of simultaneous memory address access in models that forbid it," *Technical Report* No.210, Computer Science Department, Technion, Haifa, Israel, July,1981.

[VISH82] Vishkin,U., "An optimal parallel connectivity algorithm," *IBM Research Rerort*, Yorktown Heights, 1982.

[WYLL79] Wyllie,J., *The Complexity of Parallel Computations*,

Ph.D.Thesis, Dept. of Computer Science, Cornell University, Ithaca, N.Y., 1979.

APPENDIX : Some Detailed Implementations

Algorithm DSF

```
(*To find an inverted spanning forest in an undirected graph
*)

stage 1

{ Variable declarations }
M : array[1..n,1..n] of 0..1;
$FR^+$ : array[1..2n-1,0..n-1] of 1..nlgn;
depth : array[1..2n-1] of 0..n-1;
PTR : array[1..nlgn] of 1..2n-1;
DV : array[0..lgn,1..n] of 1..n;
rootv : array[1..2n-1] of 1..n;
B : array[1..2,1..n,1..n] of 1..n;
flag : array[1..n] of 0..1;
D,C : array[1..n] of 1..n;
phase : 1..lgn; startpt : 1..2n-1;

step 1: { initialization }
     for all i:1≤i≤n pardo
                         DV[0,i]:=D[i]:=i; flag [i]:=0
                        dopar;
     for all i:1≤i≤nlgn pardo PTR[i]:=0 dopar;
     for all i:1≤i≤2n-1 pardo
                             $FR^+$[i,0]:=$FR^+$[i,1]:=0;
                             rootv[i]:=0
                            dopar;
     for all i,j:1≤i,j≤n pardo
                             B[1,i,j]:=i; B[2,i,j]:=j
                            dopar;
     phase:=0; startpt:=0;

repeat

step 2(a):
     { Pack all defined rows in each segment together}
     S:={i|flag[i]:=0 };
        {Set pointers in array PTR. second is a function
        extracting the second portion of a variable formed
        by the function concatenation in the preceeding
        step.}
     temp:=second(sort({ concat( flag[i],i)|1≤i≤n }));
     PTR[phase*n+1..(phase+1)*n] := second(sort({concat(
               temp[i],startpt+i) |1≤i≤|S|}U
               {concat(temp[i],0)| |S|<i≤n}));
     startpt:=startpt+n/2**phase;

step 2(b):
     for all i∈S pardo
```

```
                  $j_0$:=min{j|M[i,j]=1, j∈S}
                       if none then $j_0$:=i;
                  C[i]:=$j_0$;
                  $FR^+$[PTR[phase*n+i] ,0]:=phase*n+i;
                  $FR^+$[PTR[phase*n+i] ,1]:=phase*n+$j_0$
               dopar;

step 3(a):
     {Check to see if the set S can be reduced any further;
     if not, then terminate execution}
     if (for all i∈S, C[i]=i) then exit;

step 3(b):
     for all i∈S pardo if C[i]=i then flag[i]:=1 dopar;

step 4:
     for all i∈S pardo D[i]:=C[i];dopar;

step 5:
     for j:=1 step 1 until lgn do
        for all i∈S pardo C[i]:=C[C[i]] dopar;

step 6(a):
     for all i∈S pardo D[i]:=min{C[i],D[C[i]] } dopar;

step 6(b):
     for all i: 1≤i≤n pardo D[i] :=D[D[i]] dopar;

step 6(c): {Record the array D[i], 1≤i≤n }
     for all i:1≤i≤n pardo
                       if i∈S
                       then DV[phase+1,i]:= D[i]
                       elseDV[phase+1,i] :=D[DV[phase,i]]
                   dopar;

step 6(d):{ Convert the edge from the smallest-numbered
          vertex of each 1-tree-loop to a self-loop }
     for all i:D[i]=i
        pardo
          $FR^+$[PTR[phase*n+i], 1]:=$FR^+$[PTR[phase*n+i],0]
        dopar;

step 7(a):
     for all i∈S pardo
        for all j∈S : j=D[j] pardo
          Choose any $j_0$∈S such that D[$j_0$]=j and M[i,$j_0$]=1
                    if none then $j_0$:=j;
          M[i,j]:=M[i,$j_0$];
          B[1,i,j]:=B[1,i,$j_0$];
          B[2,i,j]:=B[2,i,$j_0$]
        dopar
     dopar;

step 7(b):
```

```
        for all $j \in S$ : $j=D[j]$ pardo
           for all $i \in S$ : $i=D[i]$ pardo
             Choose any $i_0 \in S$ such that $D[i_0]=i$ and $M[i_0,j]=1$
                                 if none then $i_0:=i$;
             $M[i,j]:=M[i_0,j]$;
             $B[1,i,j]:=B[1,i_0,j]$;
             $B[2,i,j]:=B[2,i_0,j]$
           dopar
        dopar;

   step 7(c):
        for all $i \in S$ pardo $M[i,i]:=0$ dopar;

   step 8:
        for all $i \in S$ pardo if $D[i] \neq i$ then $flag[i]:=1$ dopar;

   $phase:=phase+1$;

   until ($phase \geq$ lgn);



   stage 2

   step 1: { Evaluate the array $FR^+$ }
        Compute $FR^+$ and $depth[i]$ for $1 \leq i \leq 2n-1$.

   step 2:
        $phase:=phase-1$;
        { Note that at this point, each vertex $k$ left in S is
        the root of a in-tree recorded in the 'last' segment}
        for all $k$:$k \in S$ pardo
           $rootv[PTR[phase*n+k]] := k$
        dopar;
        repeat
           for all $i$: ($phase*n+1 \leq i \leq (phase+1)*n$
                        and $PTR[i] \neq 0$
                        and $FR^+[PTR[i],(n-1)-depth[i]]$
                        $\neq FR^+[PTR[i],(n-1)-depth[i]+1]$) {not
                        self-loop}
             pardo { Output all the edges except the one
                  emitting from the new root first}
                  {Denoting $FR^+[PTR[i],(n-1)-depth[i]]$ mod n
                  and $FR^+[PTR[i],(n-1)-depth[i]+1]$ mod n by
                  $v_0[i]$ and $v_1[i]$ respectively }
               if $rootv[PTR[i]]=0$ then
                  begin
                  $T[1,B[1,v_0[i],v_1[i]]]:=B[1,v_0[i],v_1[i]]$;
                  $T[2,B[1,v_0[i],v_1[i]]]:=B[2,v_0[i],v_1[i]]$;
                  end;
               {Define the roots for the next segment};
               if $phase>0$
               then $rootv[PTR[DV[phase-1,B[1,v_0[i],v_1[i]]]+$
                  $(phase-1)*n]]:=B[1,v_0[i],v_1[i]]$;
```

```
        { Reverse the edges if necessary }
        if rootv[PTR[i]]≠0
        then for all j:((n-1)-depth[i] ≤j<(n-1))
          pardo{Denoting $FR^+$[ PTR[i],j]mod n and $FR^+$[
          PTR[i],j+1]mod n by $v_0[j]$ and $v_1[j]$
          respectively}
          T[1,B[2,$v_0[j]$,$v_1[j]$] ]:=B[2,$v_0[j]$,$v_1[j]$];
          T[2,B[2,$v_0[j]$,$v_1[j]$] ]:=B[1,$v_0[j]$,$v_1[j]$];
          { Redefine the roots as well };
          if phase >0 then
           begin
            rootv[PTR[DV[phase-1, B[1,$v_0[j]$,$v_1[j]$]]
            +(phase-1)*n]]:=0;
            rootv[PTR[DV[phase-1, B[2,$v_0[j]$,$v_1[j]$]]
            +(phase-1)*n]]:=B[2,$v_0[j]$, $v_1[j]$]
          end
          dopar
     dopar;


   {Pass the roots defined in the current and previous
   segments to the next segment}
   for all i:(phase*n+1≤i≤(phase+1)*n
    and PTR[i] and rootv[PTR[i]]≠0)
     pardo
        rootv[PTR[DV[phase-1, rootv[PTR[i]]]+(phase
        -1)*n]]:=rootv[PTR[i]]
     dopar;

   phase:=phase-1;

until (phase<0);
```

```
Algorithm Bridges(M,bridge);
Input : The adjacency matrix M of a connected, undirected
        graph G(V,E);
Output: A nxn matrix bridge[1..n, 1..n] such that
        bridge[i,j]=1 iff (i,j) is a bridge in G;
        Step 1: Call inverted-spanning-forest(M,T);
Step 2: Find HLCA[i](using the method presented in Section
        8) and depth[i],∀i∈V, then computed dHLCA[i],∀i∈V;
Step 3: for all i,j: (i,j)∈VxV
          pardo bridge[i,j] :=bridge[j,i]:=0 dopar;
        for all e:e=(a,b)∈E' { e is in T }
          pardo for all i:i∈V
                  pardo
                    if $F^+$[i,(n-1)-depth[a]] =a {a is an
                    ancestor of i} and dHLCA[i] < depth[a]
                    then B[a,i]:=0
                    else B[a,i]:=1
                    dopar
```

```
            $bridge[a,b] := bridge[b,a] := \bigwedge\{ B[a,i] \mid i \in V\}$
          dopar;
```

```
Algorithm LCA($T$,LCA$^+$,$depth$);
Input : the vector, $T[1..n]$, for a directed tree $T$ such that
        $T[i]=j$ iff $j$ is the father of $i$ in $T$.
Output: The ordered pair (LCA$^+$,F$^+$);

        Step 1: Compute F$^+$;
Step 2: { Find the lowest common ancestor for $(a,b)$ where
        $(a,b) \in V' \times V'$ based on F$^+$ and binary search}
     for all $(a,b) \in V' \times V'$
        pardo
          $ptr := \lfloor n/2 \rfloor$; $\ell := 0$ $u := n-1$;
          for $t := 1$ step 1 until $\lceil \lg n \rceil + 1$ do
               begin
               if F$^+[a,ptr]$=F$^+[b,ptr]$
                    then {move left} $u := ptr$
                    else {move right} $\ell := ptr+1$;
               $ptr := \lfloor (u+1)/2 \rfloor$;
               end;
          LCA$^+[a,b] := ptr$
        dopar;
```

B30396